DESIGN AND ANALYSIS OF RECONFIGURABLE MEMORIES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ken Mai
June 2005

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark A. Horowitz
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Bruce A. Wooley

Approved for the University Committee on Graduate Studies:

Abstract

Decades of technology scaling have made available an unprecedented amount of computational power from today's integrated circuits. However, continued process scaling has come at the price of ever more exotic process technologies and complex designs. Due these difficulties, the non-recurring engineering cost of custom ASIC development is growing, making ASICs economically infeasible for all but the highest volume parts. However, future applications will require more efficient, high-performance computation than general purpose processors can provide. One promising approach to breaking this impasse is to use reconfigurable architectures that keep the low non-recurring engineering costs of general purpose silicon, yet still provide efficiency and performance near that of custom ASICs. While there is a large body of work on designing reconfigurable computation, reconfigurable memory systems have been largely ignored. In this work, we examine how to add reconfigurability to the memory system.

Looking closely at how memory is used in modern digital systems, we recognized that the most common memory structures, such as caches, FIFOs, and scratchpads, use very similar memory building blocks. By adding a few meta-data bits and a small amount of peripheral logic to a basic SRAM array, we designed a reconfigurable memory mat that could form the core of a many common memory structures. Adding a flexible interconnection network between the mats and the computation facilitates aggregation of the mats into larger, more complex memory structures.

To evaluate our design, we implemented a prototype reconfigurable memory testchip in a 0.18μm CMOS technology. The testchip operates at 1.1GHz at the nominal 1.8V Vdd and room temperature. The prototype uses a 16kb SRAM mat and achieved area and power overheads of 32% and 23% of the totals respectively. Our projections show that the

reconfiguration overheads can be reduced to below 15% of the area and below 10% of the power by using larger capacity SRAMs. The testchip shows that we can build a generic reconfigurable memory block that can form the basis of many different memory structures, while maintaining high-performance and low power.

Acknowledgements

One of the benefits of having an extremely long graduate student career is that I have had the opportunity to interact with and learn from many wonderful people. Their contributions to this work have been invaluable.

I could not have asked for a better advisor than Mark Horowitz. His technical depth and breadth, ability to bore to the heart of a problem, steadfast refusal to cut corners, scientific integrity, dedication and accessibility to his students, and patience made him a great advisor and role-model. I would like to thank him for giving me the opportunity to work with him.

My reading committee members, Kunle Olukotun and Bruce Wooley, and my orals committee members, Christos Kozyrakis and Boris Murmann, provided insightful comments on my research.

Charlie Orgish, Joe Little, and Jason Conroy kept the computing infrastructure running smoothly. The support staff, notably Darlene Hadding, Terry West, Deborah Harber, Lindsay Brustin, Taru Fisher, Penny Chumley, Teresa Lynn, and Ann Guerra, provided administrative assistance.

Fujitsu Labs, Philips, and DARPA offered their financial support.

The Horowitz research group has consistently been an engaging, friendly, and helpful environment. Ron Ho has been my near-constant collaborator and a good friend. This work has benefited greatly from his help and expertise. Also deserving of special note are Guyeon Wei, Dan Weinlader, Birdy Amrutur, and Elad Alon who have not only been good colleagues, but also good friends. At times our forays into non-research related topics may have lengthened our time in graduate school, but their friendship most certainly made the time more enjoyable. My officemates, Toshi Mori, Junji Ogawa, and Sam Palermo stoically tolerated me and the mountains of papers.

My parents and brother Glenn offered their encouragement and support.

My wife Kaarin’s graduate student career overlapped with mine, but despite her own trials and travails, she has always been encouraging, empathetic, and supportive.

Finally, I would like to thank our cats, Simon and Max, who somehow always knew when a good hairball was the proper stress reliever.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Decades of technology scaling have made available an unprecedented amount of computational power from today's integrated circuits. This has opened up new application areas such as computational biology and ubiquitous computing, where previously computation was either not powerful enough or not efficient enough to attack the problem. These types of new application areas will keep up the demand for ever more efficient, high-performance computation.

However, continued process scaling has come at the price of ever more exotic process technologies and complex designs. Designers face increasing difficulties such as interconnect delay, device mismatch, leakage, power density constraints, and complexity management. Due these difficulties, the cost of custom ASIC development is increasing at an alarming rate. Within the next few process generations, the cost of mask sets will exceed $1 million [1], and the design costs will reach into the tens of millions of dollars [2].

For system designers, the increasingly high cost of custom ASIC development means that the highest performance, highest efficiency solutions for their applications are becoming economically infeasible. Broadly speaking, system designer can choose to use off-the-shelf parts or custom design an ASIC for their application. The off-the-shelf parts have low non-recurring-engineering costs, yet lag the custom ASICs in performance and efficiency, sometimes by orders of magnitude [3]. Further, the off-the-shelf, general purpose processor solutions are losing steam, running out of available parallelism and encountering

architectural scaling problems [4]. While some applications can still use off-the-self processor and memory solutions, a growing number of applications need better performance and efficiency than off-the-shelf components can provide, but cannot economically justify custom ASIC development.

These applications have stimulated a growing interest in reconfigurable computing solutions. Field programmable gate arrays (FPGAs) [5][6] have been commercially available for quite some time and have been gaining in popularity as fast prototyping and emulation platforms. Additionally, a number of academic and industry efforts have combined a general purpose processor with an FPGA fabric [7][8][9] for both ease of programming and reconfigurability. Further, some next-generation computing architecture projects have explored architectures that can exploit multiple types of application parallelism [10][11] and even entirely polymorphic computing architectures [12][13][14].

These solutions bridge the gap between custom ASICs and general purpose solutions by allowing the user to customize a pre-fabricated reconfigurable component. Thus the non-recurring-engineering costs remain low, while the performance and efficiency are better than those of off-the-shelf, general purpose solutions [4][15][16]. However, these solutions still lag custom ASICs in performance and efficiency due to limitations in their reconfigurability and overheads associated with that reconfigurability [3][15]. This illustrates the fundamental design trade-off between efficiency and flexibility: the more general-purpose a system, the less efficient it is at a specified task; a highly directed design can be extremely efficient at the target application, but is very inefficient at other tasks.

While there is a large body of work on how to design reconfigurable compute units, the memory systems of such designs have been largely ignored. Many systems with reconfigurable computation, have entirely hardwired memory systems [12][14]. FPGAs have only recently added small, somewhat reconfigurable memory blocks in the computing fabric. However, the memory system plays a critical role in determining the performance, power, and cost of modern machines [17][18]. Just as every application has unique compute requirements, every application has unique memory access patterns which perform optimally using different memory paradigms. A memory system that can be reconfigured to match the application memory access pattern requirements can significantly boost both the performance and efficiency of the system. Building such a memory system is the goal of this

thesis.

1.1 Organization

To set the stage for our discussion of reconfigurable memory, we begin in Chapter 2 by reviewing contemporary SRAM design techniques. These techniques include: array partitioning; post-charged decoder logic; datapath circuit design; control and clocking using replica timing; and efficient data and address signaling and distribution. We then examine how these highly optimal SRAM blocks are used to form caches, FIFOs, and scratchpads in modern memory systems.

Chapter 3 explores motivating factors for reconfigurable memory and the architecture of our proposed memory system. The architecture is driven from below by memory circuit design considerations and driven from above by the required architectural flexibility. We also detail the design and implementation of the base reconfigurable memory block called a *memory mat*.

We continue the description of the proposed memory system in Chapter 4 by detailing the interconnection networks between the memory mats and between the mats and the computation. Each network has unique communication requirements and characteristics, and we tailor the implementations to efficiently meet those needs.

To explore the feasibility and overheads of our architecture, we designed and implemented a prototype reconfigurable memory testchip. Chapter 5 describes the testchip implementation and measured results. From these results, we extrapolate to larger, more complex implementations of the proposed reconfigurable memory system.

Chapter 2

SRAM Design

In today's processors and ASICs, one of the most voracious consumers of die area and device count is the on-die memory. The predominant memory technology used on-die with computation is 6T SRAM. While embedded DRAM has appeared in a number of academic and commercial designs, such as the Berkeley IRAM project [19], gaming consoles [20][21], and the Mosys 1-T "SRAM" memory block [22], it has not been widely used due to the additional cost of a merged logic-DRAM process and higher design complexity. A 6T SRAM memory, however, can be manufactured in a standard logic process and continues to offer high-performance at a reasonable density and power dissipation. SRAMs will likely remain the dominant on-die memory technology, as SRAM cells have already been demonstrated down to the 32nm technology node [23].

A 6T SRAM cell uses a pair of cross-coupled inverters as its bi-stable storage element with two additional NMOS devices for read and write access (Figure 2.1). The cells are aggregated into cell arrays to share the decoding and I/O logic (Figure 2.2). On a read, the decoder raises the wordline (*WL*) of the desired word. The bitlines (*BL* and *BL_b*) have been precharged to a reference voltage, and the cell drives a differential current onto the bitlines according to the stored value. The cell current is relatively weak for the bitline capacitance, so to speed the read operation, a sense amplifier in the I/O logic amplifies the bitline differential voltage to produce a full swing logic value. On a write, the write driver in the I/O logic places the write data onto the bitlines as full-rail signals. The decoder again raises the wordline of the accessed word, and the cells store the new data values. A number

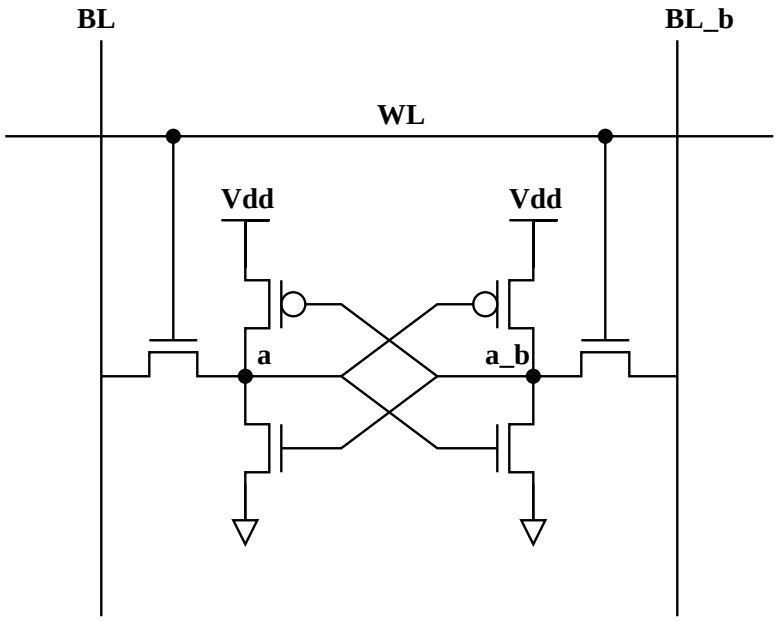

Figure 2.1: 6-transistor SRAM cell

of cell columns can share I/O circuitry via the column multiplexor. There are a number of ways in which designers have optimized the architecture and circuit design of SRAMs to improve the performance and energy efficiency. In this chapter, we will review a few of the most prevalent and significant optimizations.

2.1 Array Partitioning

While the simple monolithic cell array architecture in Figure 2.2 is appropriate for small memories on the order of a few Kbytes, for larger memories, designers partition the cell array for better performance and energy efficiency [24][25][26]. For partitioned arrays, we will use the same terminology as Amrutur [24] to describe the partitioning. An SRAM is divided into *nm* macros, each of which is accessed simultaneously. Every macro operates independently, with the exception of possibly sharing a portion of the decoder with the other macros. Each macro contains a portion of the accessed word called the sub-word. Each macro is again divided into a number of blocks. The requested sub-word is contained

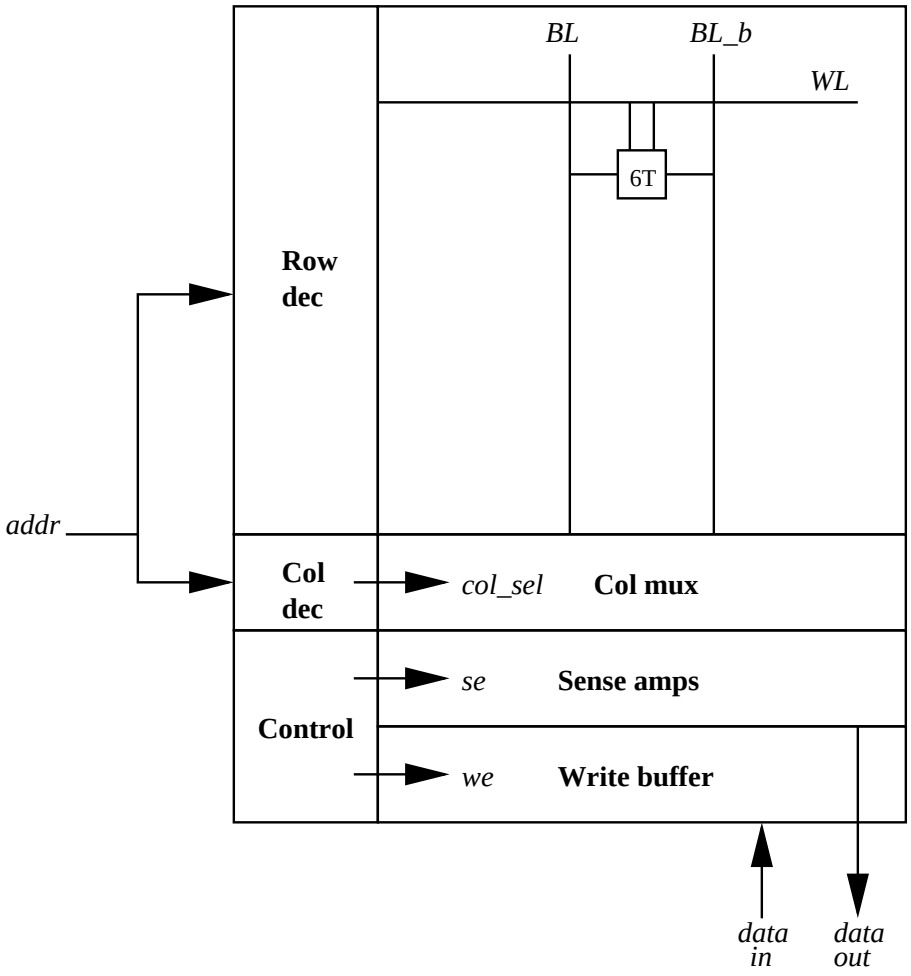

Figure 2.2: SRAM block diagram

entirely in the block, which we define as an array of cells that shares local wordline drivers and bitline I/O circuits.[1] Each block has *bw* cells in a row, and *bh* cells in a column. The bitlines run vertically, and the wordlines run horizontally.

Figure 2.3 shows an example partitioning of a 512Kb SRAM. The SRAM has 4 macros, each with 8 16Kb blocks. The blocks are 128 rows by 128 columns (*i.e. bh* = 128 and *bw* = 128) and have an access width of 16b. The 16b access width requires each block to perform 8:1 column multiplexing.

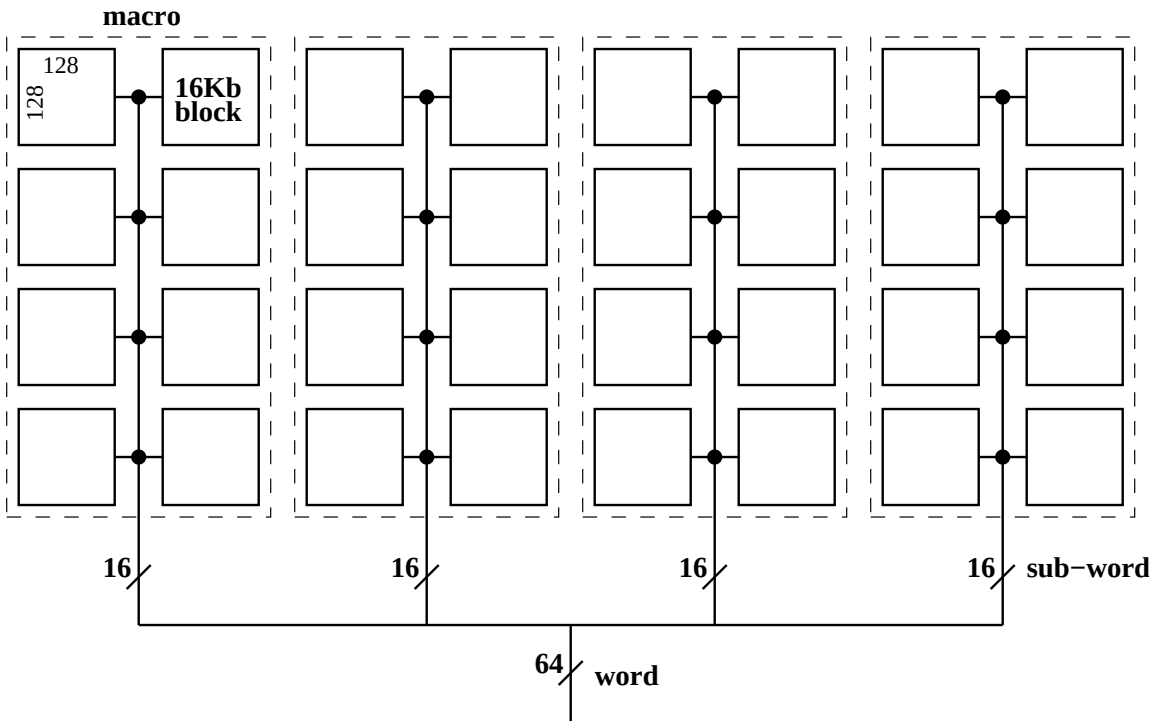

Figure 2.3: Partitioned 512Kb SRAM array using 4 macros each with 8 16Kb blocks

In a partitioned SRAM only a portion of the array is activated every access. By using hierarchical wordline decoding [27][28] and hierarchical bitline architectures [29], designers can keep the lines short and avoid significant wire RC delays. The shorter, lower capacitance lines and partial activation of the array serve to reduce the energy per access.

[1] A block can be though of as logically a monolithic cell array, but in practice it does not have to be. For example, a block could place the decoder in the middle of the cell array to reduce the wordline RC delay, and thus actually contain two monolithic cell arrays.

Partitioning does, however, decrease the area efficiency of the SRAM, and designers must trade that off against the improved energy and performance. Eventually this increase in the area offsets the gains from a smaller block and is detrimental to the overall performance and energy dissipation of the memory [24]. The optimal block size depends on the optimization goal (*e.g.* delay, energy, area, energy-delay), SRAM architecture, circuit style, and technology, but typically is in the 16KB to 128KB range and is not a strong function of process technology [25][30][31][24].

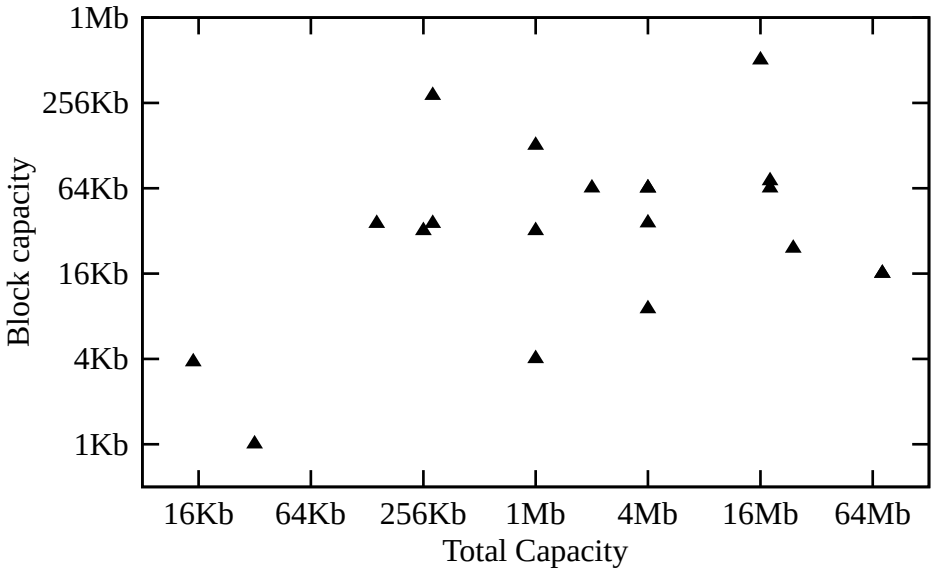

Figure 2.4: SRAM block size survey (see Appendix A)

Figure 2.4 shows a scatter plot of the block sizes for a number of recent SRAM designs. The detailed characteristics and full citations of the SRAMs can be found in Appendix A. With the exception of a few outliers, most of the SRAMs have partition sizes that lie between 16Kb and 128Kb. This bears out the conclusions of both Amrutur [24] and Evans [25] that the optimal energy-delay partition size falls within this range. The SRAMs plotted were fabricated in processes that span the 90nm to 0.65μm technology generations. The large block outliers are SRAMs that were optimized for area efficiency. The small partition outliers are SRAMs that were optimized for extreme low power or high performance operation.

For a partitioned SRAM, we can break the memory access down into four phases: request transport, block decode, block datapath, and reply transport. In the *request transport* phase, we send the the address and data from the global inputs to the requested block. A portion of the decode may occur during this phase in the selection of which block to access [27][28]. The *block decode* phase is the local decode from the block address input to the local wordline assertion. The *block datapath* phase is from the wordline assertion through the cell driving the bitlines to the output of the local sense amplifier or the cell writing in the data on the bitlines. The *reply transport* phase is from the output of the local sense amplifier to the global data output. This phase is only necessary if the request returns data. Other researchers have chosen to break a memory access into two parts, calling our first two stages *decode* and last two *datapath*, but as we will see in Chapter 3, our four segment division is logically cleaner for our reconfigurable memory design.

2.2 Decoder Design

The decoder logic can be thought of as a series of high-fanin AND gates with a very low activity factor. For a memory with 2^n words, the decoder is logically 2^n n-input AND gates, designed such that only one AND-gate fires (*i.e.* raises its wordline) for a given n-bit address input. The decoders are usually hierarchical [27][28] sharing pre-decoder gates for common boolean terms. The address inputs are typically clustered into pre-decode groups of three to four bits for pre-decoding. The pre-decoder outputs are then combined in the global and local row decoder gates to generate the block wordline.

For a fast decoder, we would like to use a low logical effort [32] logic family like precharged domino logic. However, while domino logic would provide high performance, the precharge phase is excessively wasteful in energy for decoders, because decoders have an inherently low activity factor. On each access, only a small percentage of the gates in the decoder fire. But in precharged logic, the precharge control signal goes to all gates in the decoder and drives the gate capacitance of all the reset devices, regardless of whether they have discharged or not. This is excessively wasteful in energy, because we only needed to reset the gates that fired. Ideally, we would only reset those gates, because all other gates would not require resetting.

To retain the high-performance of precharged logic, but achieve low-power, designers instead use post-charged, self-resetting logic styles in the decoder [33][34]. As the name implies, a short delay after a self-resetting gate asserts its output, it resets itself . The output is a pulse whose pulsewidth is set by the self-reset delay. The pulse-mode nature of these gates allows us to carefully control the pulsewidth of the wordline which is critical for decreasing the read power [35]. For a read, the wordline pulsewidth determines how long the accessed cells drive the bitlines and thus the read bitline swing and bitline energy dissipation. Generally, the wordline pulsewidth is set to be just long enough to generate the bitline differential voltage needed to overcome offsets in the sense amplifiers. Section 2.3 discusses this further.

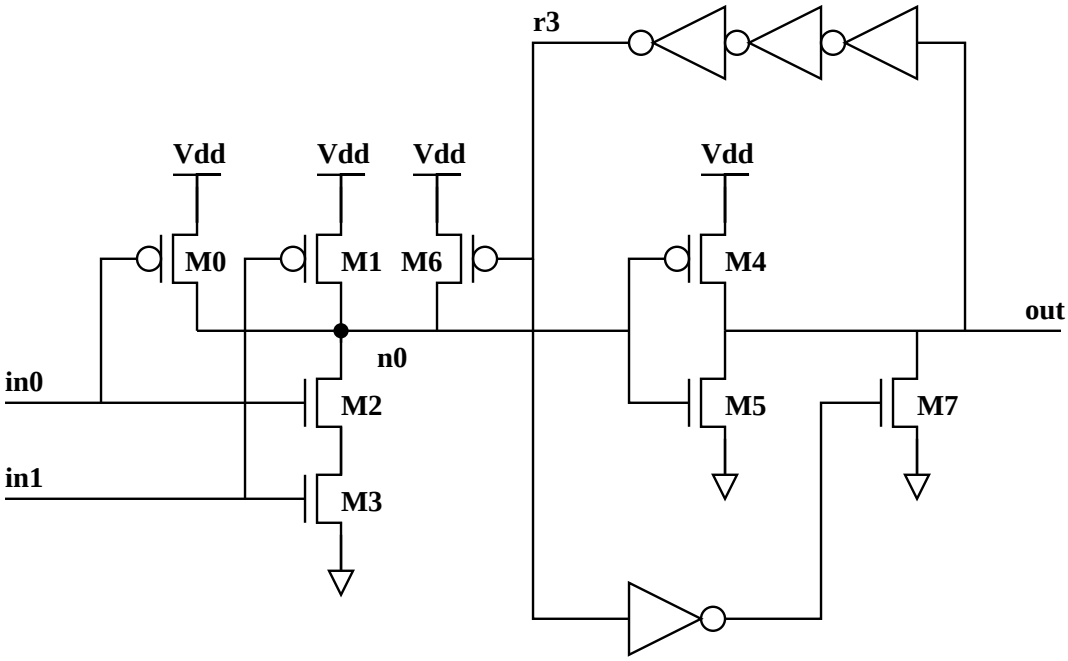

Figure 2.5: Self-resetting, 2-input, AND gate

Figure 2.5 shows the circuit schematic for a simple, self-resetting, 2-input, AND gate. Figure 2.6 shows the timing diagram for the gate. When both inputs *in0* and *in1* pulse high, *M2* and *M3* pull node *n0* low. *N0* going low pulls the output *out* high through the *M4*/*M5* inverter. After the three inverter delay, *r3* goes low turning on the reset device *M6*. *M6* pulls *n0* high, back to its quiescent state. We could wait for the *M4*/*M5* inverter to pull *out* low

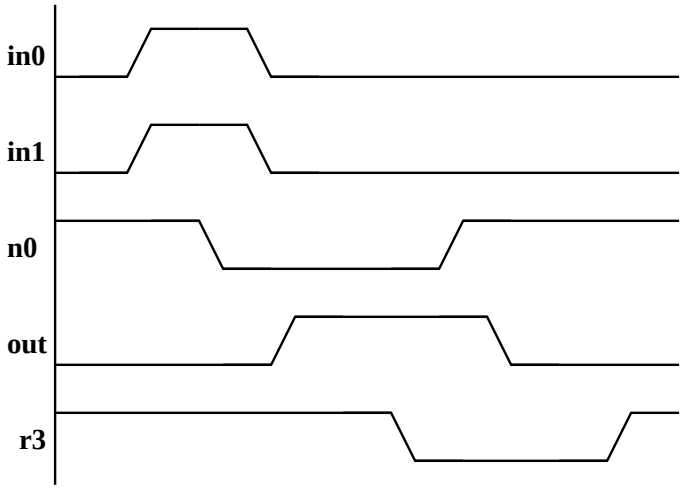

Figure 2.6: Self-resetting, 2-input, AND gate timing diagram

again, but we can speed the *out* transition edge using an explicit reset device *M7*. Both *in0* and *in1* are assumed to be pulses and must be low by the time *r3* goes low, turning on *M6*, to avoid a drive fight between *M2*/*M3* and *M6*. The gate that generates *in0* and *in1* must carefully control their pulsewidth to avoid this drive fight. Other designs for self-resetting gates can operate correctly without this restriction on the inputs [31].

We can skew the NAND gate formed by *M0*-*M3* for a fast assert edge, *n0* pull-down, and the *M4*/*M5* inverter can be skewed for a fast *out* pull-up. This speeds the assert transition of the output, but slows the reset transition. By adding the explicit reset devices *M6* and *M7* we can also have a sharp reset edge (*i.e. out* pull-down). Thus we can precisely control the pulsewidth of the output, which is key for controlling the read bitline swing and minimizing the read power [35].

Using self-resetting gates allows the decoder to achieve both high-performance and low-power. In 2001, Amrutur and Horowitz explored the design space and found that the optimal decoding structure consists of a mixture of different types of dynamic AND gates [34]. Their optimal decoder uses an initial stage of clocked Nambu OR-gates [36], followed by self-resetting, source-driven NAND gates [37][38].

The Nambu OR-gate allows us to use a fast dynamic NOR gate topology, while keeping the power dissipation low. If we used a traditional dynamic NOR gate in the predecoder,

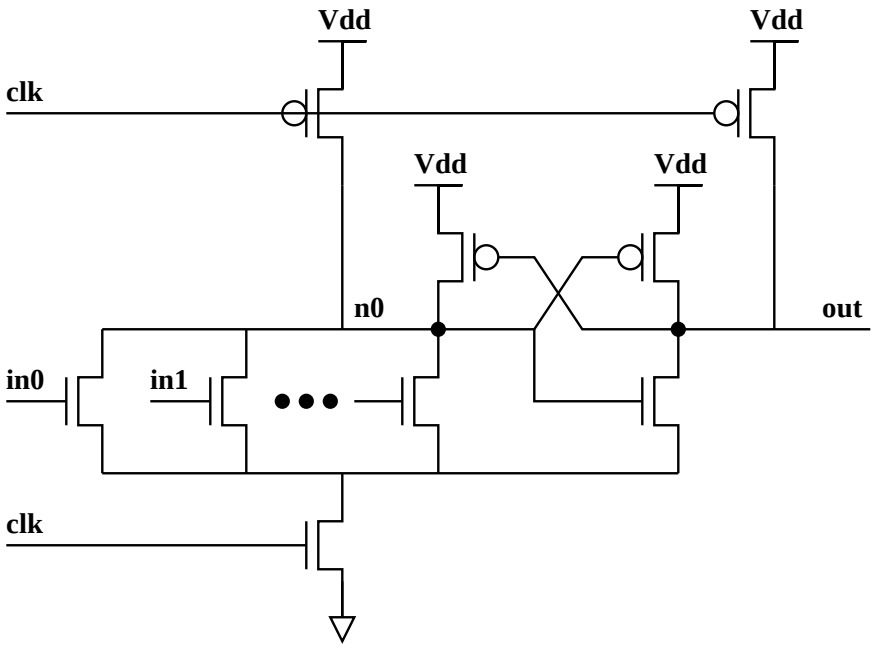

Figure 2.7: Nambu OR-gate

then for each predecoder, all but one of the outputs would transition. Only the gate that had all its inputs low would not transition. While this would be a fast circuit topology, it would dissipate a lot of power. The Nambu OR-gate retains the fast dynamic NOR topology, but avoids high power dissipation, by directly cascading two dynamic stages as shown in Figure 2.7. Correct operation of the gate relies on the dynamic NOR evaluating faster than the following dynamic inverter and requires careful sizing and simulation.

A source driven NAND gate (Figure 2.8) implements the NAND function using a circuit topology that only has a single NMOS device in the pull-down stack. One of the inputs *in_n* drives the source of the NMOS device rather than a transistor gate. *In_n* is asserted when it is low, and *in_p* is asserted when it is high. Thus, when both input are asserted the NMOS device has a full Vdd across its Vgs and is fully on. This gate can achieve delay near that of an inverter, while still performing the NAND function. This type of gate has also be used with half-swing inputs to reduce power dissipation if low-Vt devices are available [39][40].

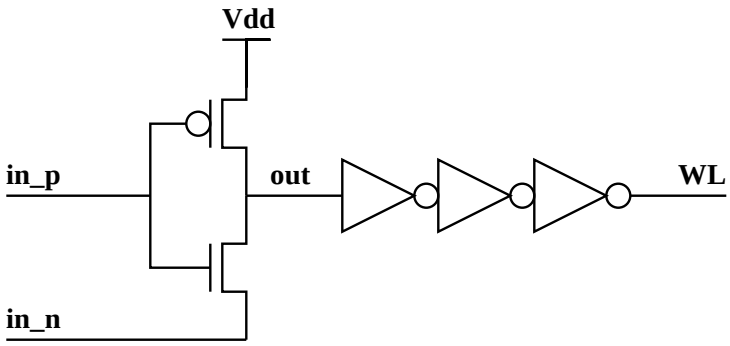

Figure 2.8: 2-input source driven NAND gate

2.3 Datapath Design

The block datapath phase is from the assertion of the wordlines through to the cell I/O circuits. The cell I/O consists of the read sense amplifiers, the write drivers, and the bitline reset devices. Both the sense amplifiers and write drivers may have a column mux between them and the bitlines if they are shared among multiple columns.

2.3.1 Sense Amplifier

On a read access, the cells in the selected row drive a differential current on to the bitlines based on their stored values. The bitlines have been previously equalized and reset to a reference voltage by the bitline reset devices. While we could simply wait until the bitlines have slewed full-rail to a digital logic value, to save power and reduce the read delay, most design use sense amplifiers to sense the the bitlines as soon as their differential voltage has reached approximately 100mV, enough to overcome the built-in sense amplifier offsets [35]. When the sense amplifiers are enabled, the wordlines are also shut off to prevent additional, unnecessary swing of the bitlines and wasted power.

To further reduce power dissipation, designers have eschewed static sense amplifiers, in favor of clocked designs. Figure 2.9 shows a commonly used latch-type sense amplifier. Clocked sense amplifiers have a sense enable signal *se* that triggers the sensing. The figure shows a 2:1 column multiplex using only PMOS devices. This assumes that the bitline reference level is Vdd, so we only need PMOS devices for the passgate column mux. Some

lower reference levels (*e.g.* Vdd/2) may require a full transmission gate for the column mux.

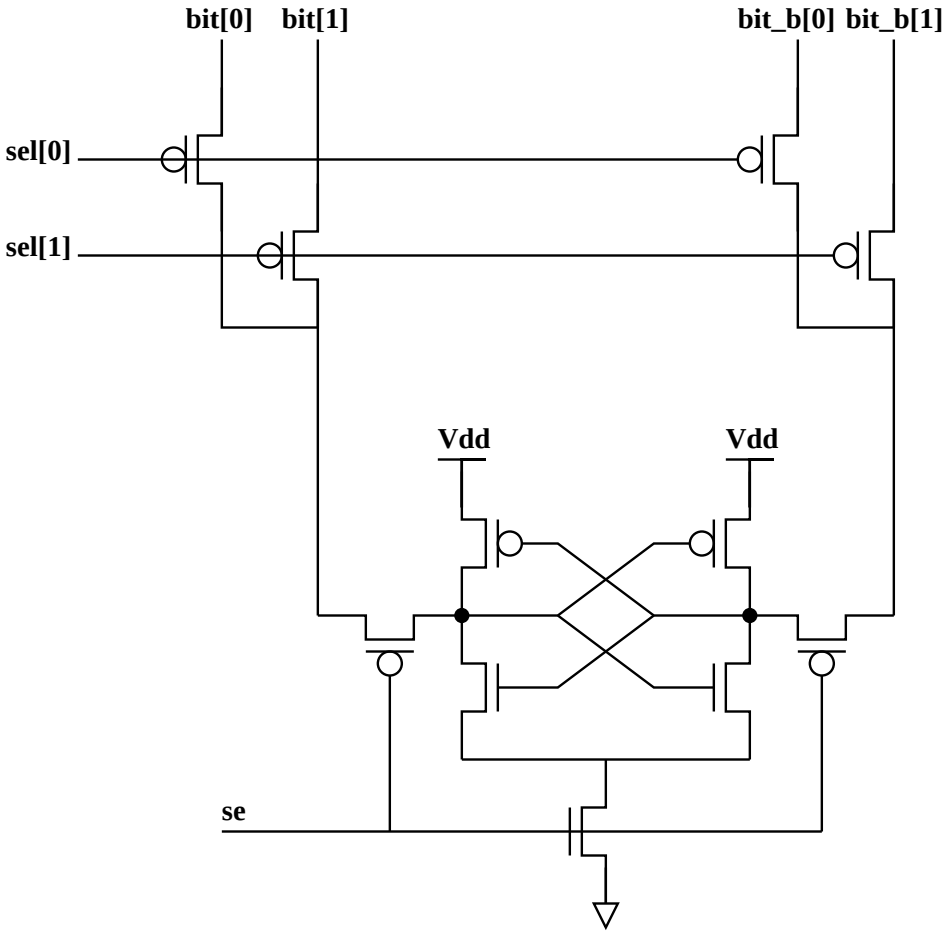

Figure 2.9: Latch-style sense amplifier

There are many different clocked sense amplifier circuits, including both voltage mode and current mode designs. SRAMs that are willing to swing the bitlines full-rail on a read can use skewed inverters for simple single-ended voltage mode sensing.

2.3.2 Write Driver

On a write access, the write driver sets the bitlines to the input data value. After the decoder asserts the selected wordline, the cell stores the data value on the bitlines. For a successful write, the bitlines must typically swing full-rail or nearly so. There has been some

work on low-swing writes, but the proposed techniques have not been widely adopted [41][42][39][40][31][43][44][45]. The write enable control signal *we* controls when the write driver asserts the input data value on the bitlines. We use an NMOS only passgate mux for the 2:1 column multiplexor, because the write driver pulls the bitline to ground but relies on the bitline reset to keep the bitline that remains high near Vdd. The write drive as shown has three NMOS devices in a series stack. For a low effective resistance, this would require the NMOS devices to be quite large. To reduce the stack height, we can pre-compute the AND of *we* and *data* and *data_b*, and only have two devices in the stack (Figure 2.11).

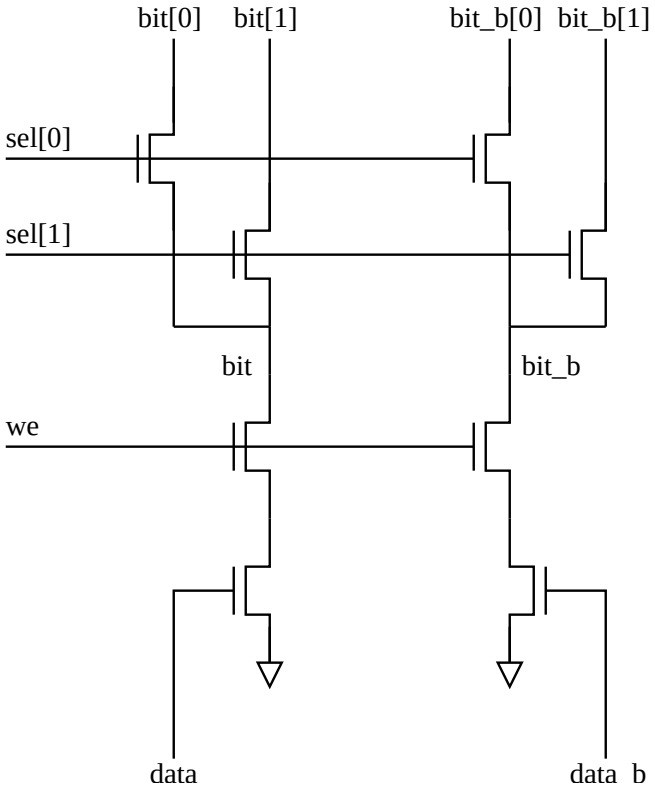

Figure 2.10: Write driver with 2:1 NMOS-only column mux

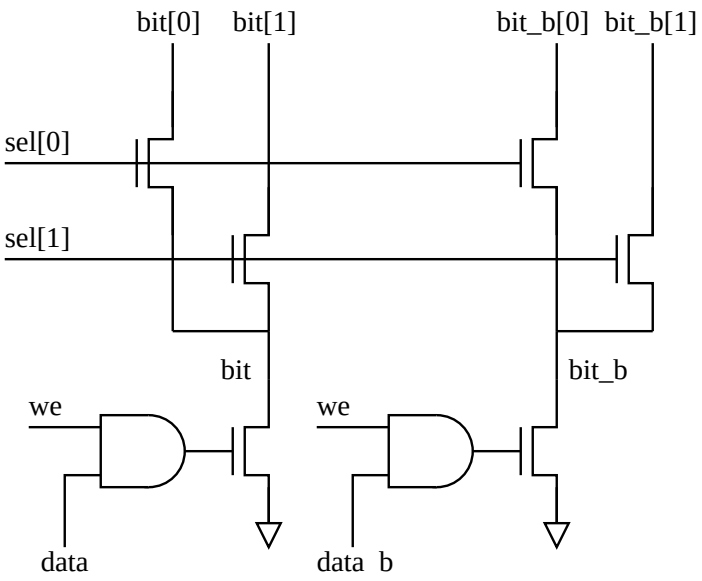

Figure 2.11: Write driver with reduced NMOS stack height

2.3.3 Bitline Reset

The bitlines sit at a pre-determined reference voltage level when the block is not active. On a read or write the bitlines deviate from that reference voltage, and the bitline reset circuits restore the bitlines to the reference voltage level after an operation. The bitline loads can either be static or clocked. Static loads do not require any complex control or clocking, but dissipate static power anytime the bitlines deviate from the reference voltage. Most modern designs use clocked reset circuits like the ones shown in Figure 2.12. The reset devices pull the bitlines to the reference voltage, typically Vdd, when the bitline reset control signal *bl_rst_b* is asserted (low). The shorting device ensures that *bit* and *bit_b* reset to the same voltage level which is especially important for reads.

There have been some designs that use a lower bitline reference voltage, usually Vdd-Vt or Vdd/2, to reduce the bitline energy [39][40] and to reduce leakage onto the bitlines from nominally unselected cells [46], but again, these techniques have not been widely adopted.

The amount of bitline reset required is typically quite different for reads and writes. On a read, the bitlines only dip about 100mV from the Vdd reference level, but on a write, one

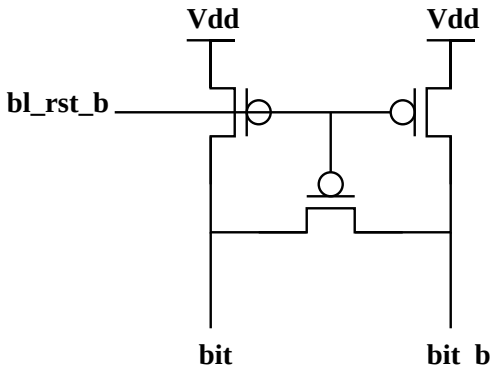

Figure 2.12: Bitline reset circuit

of the bitlines is driven to Gnd. Given the very different swings that need to be corrected, some designs have separate read and write bitline reset circuits. The write reset circuits use larger devices, because one of the bitlines must be brought all the way from Gnd to Vdd. Figure 2.13 shows a split reset circuit using larger recovery devices for the write reset which are enabled by *wr_bl_reset_b*.

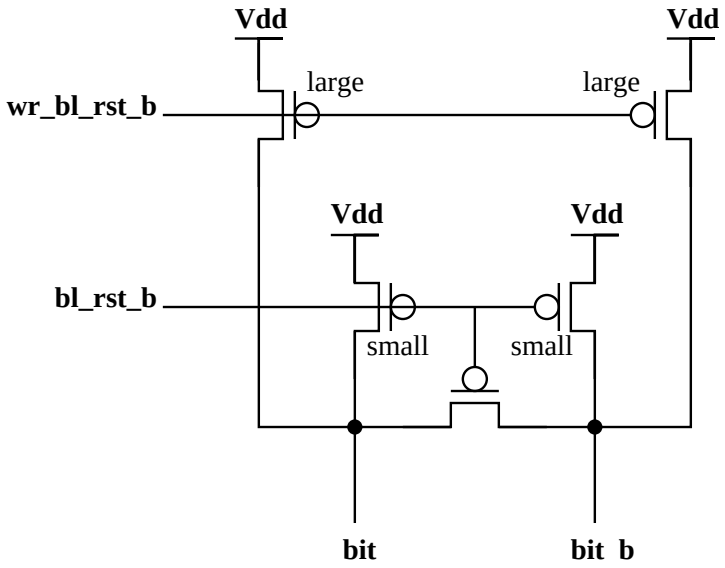

Figure 2.13: Bitline reset circuit with split read and write reset

Designers can also add static or pseudo-static keeper devices on the bitlines to mitigate

the effects of leakage and crosstalk. A static load can be built using a grounded gate PMOS device with the source connected to Vdd and the drain to the bitline. A pseudo-static load, similar to those used in dynamic logic gates, consists of an inverter driving a PMOS device as shown in Figure 2.14. The advantage of this technique is that the keeper PMOS device shuts itself off during writes, thus avoiding excessive power dissipation. However, it may be difficult to fit the extra devices needed for the pseudo-static load in the small horizontal cell pitch.[2] Some designs have used more complex techniques to actively compensate for the leakage onto the bitline due to nominally off cells [48][49][50].

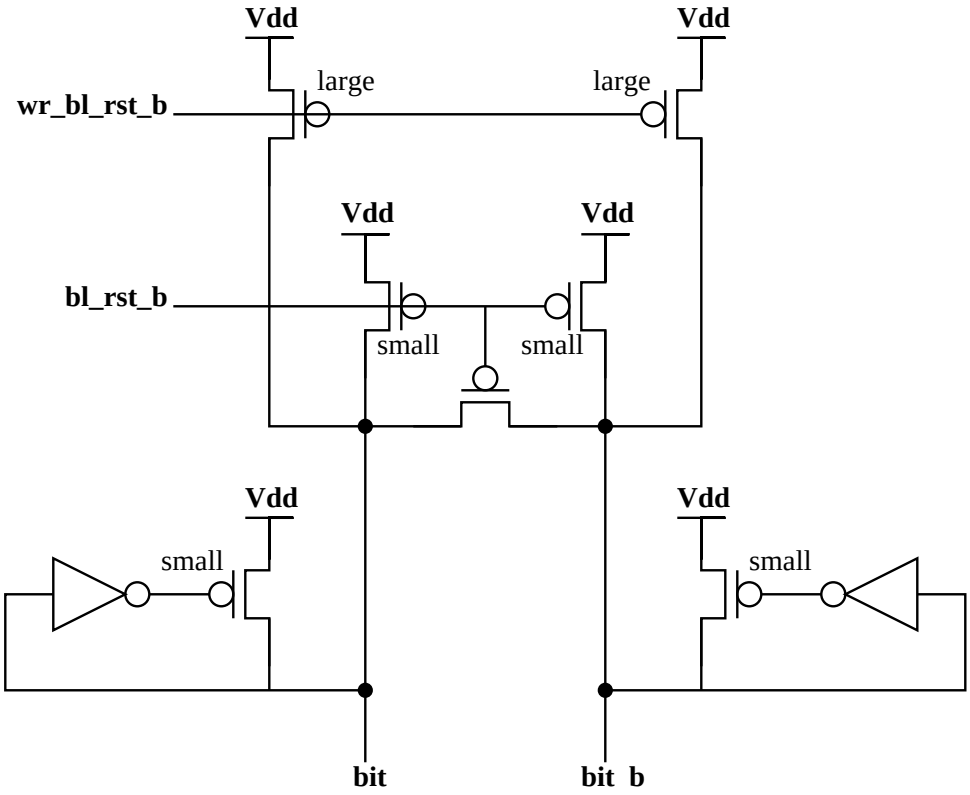

Figure 2.14: Bitline reset circuit with pseudo-static keeper circuit

[2]This may become less of an issue as designers move to a short, wide cell layout for shorter bitlines and better manufacturability [47].

Table 2.1: SRAM control signals

signal	name	unit	operation	description
se	sense enable	sense amplifier	read	enables sense of bitlines
we	write enable	write driver	write	enables drive of write data onto bitlines
bl_rst	bitline reset	bitline load	both	turns on bitline reset devices
wl	wordline	cells	both	turns on cell access devices

2.3.4 Clocking and Control

An SRAM has a number of key control signals whose timing relationships must be tightly controlled to maintain correct operation. There are additional timing relationships that can be maintained for high-performance, low-power operation. Table 2.1 lists some of the key control signals.

We can generate the control signals either by delay matching or clock selection. The clock selection method requires that there be a number of finely spaced clock signals available. Either by simulation-based dead reckoning at design time or by a training sequence, we generate the control signals from the clock edges that correspond most closely with the ideal control signal edges [51].

The delay matching method generates the control signals based on replica timing paths that match the delay of the SRAM access path [52][31]. Matching the delay of elements in the decoder is relatively easy, because the decoder consists of logic gates and buffers. By using delay elements that are made up of identical gates or using logical effort and simulation, a delay element can be made that matches the decoder delay reasonably well across process, voltage, and temperature variations [31].

However, during a read, there is an element in the SRAM access path that is not a logic gate, namely the cell driving the bitline. Matching the delay of the cell driving the bitline is critical for accurately generating the sense enable signal that activates the sense amplifiers. If we assert the sense enable signal too early, there may not be a sufficient differential voltage on the bitlines to overcome offsets in the sense amplifier, and the sense amplifier may latch-in incorrect data. If we fire the sense enable too late, the SRAM will still operate

correctly, but the performance will not be optimal. Amrutur [35] showed that a logic delay chain does not track the read bitline delay very well over a range of process, voltage, and temperature variations.

However, a replica bitline that mimics the delay of the actual bitline can track the actual bitline delay well over process, voltage, and temperature variations. To mimic the actual bitline as accurately as possible, we use a driver cell on the replica bitline that is identical to a real SRAM cell, except that it has a hardwired stored value. Unlike the actual bitline that only swings approximately 100mV, the replica bitline must generate a full digital logic level output level. We can accomplish this by either a capacitance ratioed or current ratioed replica bitline [35]. A capacitance ratioed replica bitline uses a single driver cell and a replica bitline that is a fraction of the length and capacitance of the real bitline. A current ratioed replica bitline uses a full length and capacitance bitline, but use multiple driver cells. The exact length of the replica bitline in the capacitance ratioed case or the exact number of driver cells in the current ratioed case can be fine tuned in simulation to match the actual bitline delay. Figure 2.15 illustrates the two types of replica bitlines. The capacitance ratioed replica bitline uses a rbl k times shorter than the real bitline, and the current ratioed replica bitline uses j times the drive strength as a regular cell.

While a replica bitline does track the actual bitline much better than a chain of logic gates, the tracking is not perfect. Additionally, despite careful simulation-based tuning, the replica path and the actual path may differ due to random device mismatch, inexact modeling, or local voltage and temperature variations. To ensure that the memory will operate correctly under worst-case conditions, designers pad the the replica path by between 10% and 30% extra delay. This extra time may end up being wasted time if the actual path is faster than the replica path, but ensures correct operation under worst-case conditions.

2.4 Transport Design

The request and reply transport phases transfer the address and data to and from the memory block. These phases of the memory access can account for the majority of the delay and energy in large, partitioned memories [25][31]. Designers use a number of techniques to reduce the delay and power of the transport phases.

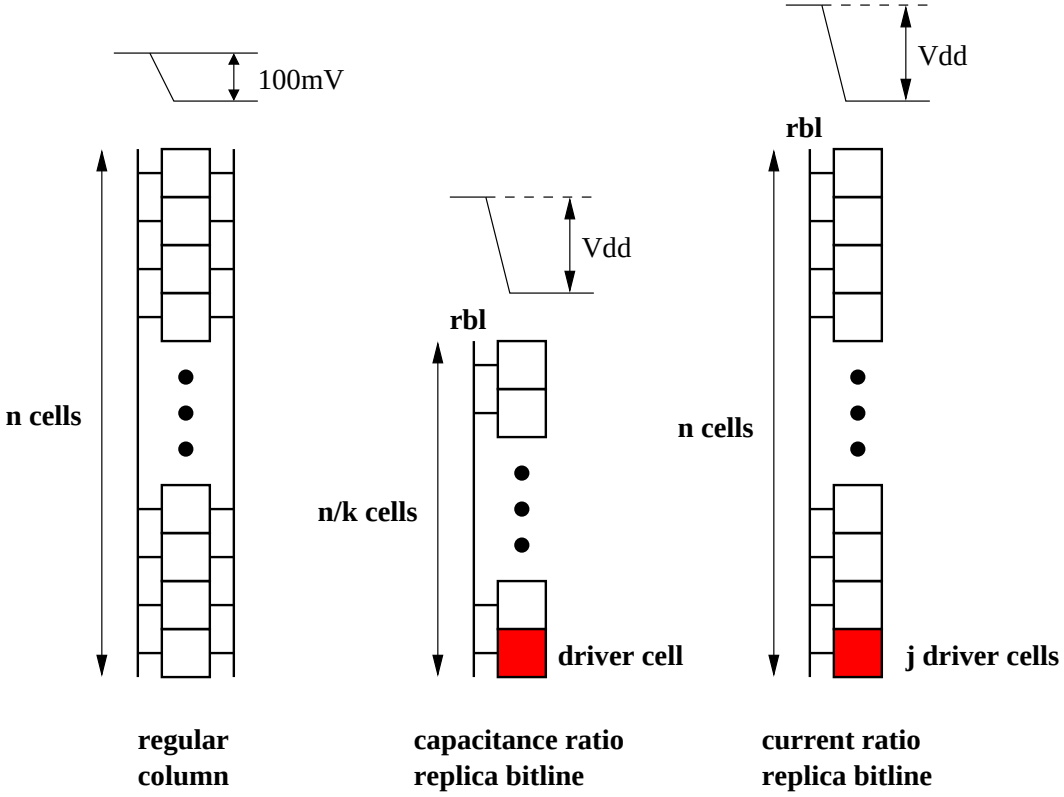

Figure 2.15: Capacitance and current ratioed replica bitlines

In his SRAM design space study, Evans concludes that "multiplexed address line" designs that route address and data to the blocks are optimal for low energy and high performance [25][53]. These designs perform a portion of the decode in the interconnect in selecting which blocks should be accessed. Broadcasting the request and reply data and address to all blocks is excessively wasteful of energy.

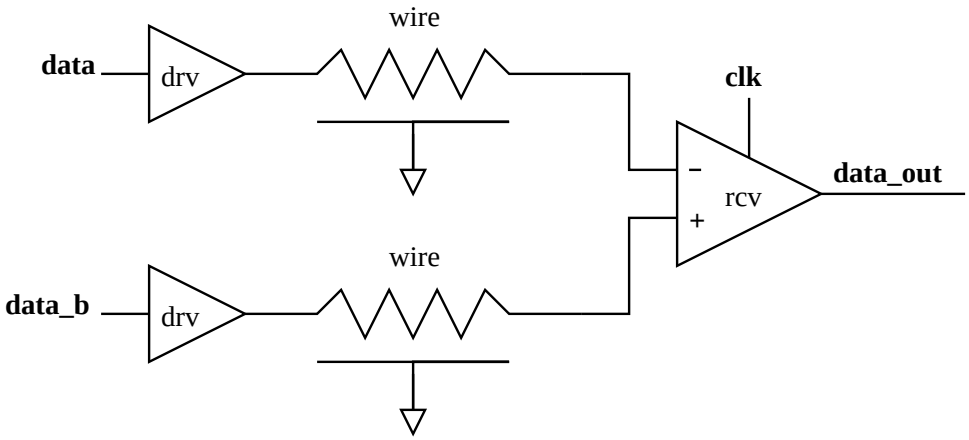

Figure 2.16: Example differential low-swing interconnect

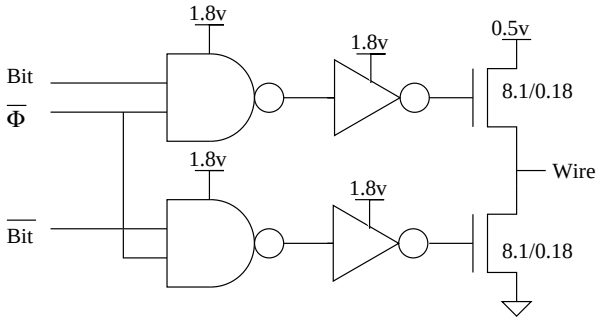

Figure 2.17: Example low-swing driver, sized for a 0.18μm technology[54]

To reduce the energy of the transport further many memories use low-swing signaling [31][39][40]. Low-swing schemes can also improve the performance of the interconnect over full-swing methods [55][54]. There are a number of low-swing signaling techniques that have been used in memories and other types of designs [55][56]. Figures 2.16-2.18

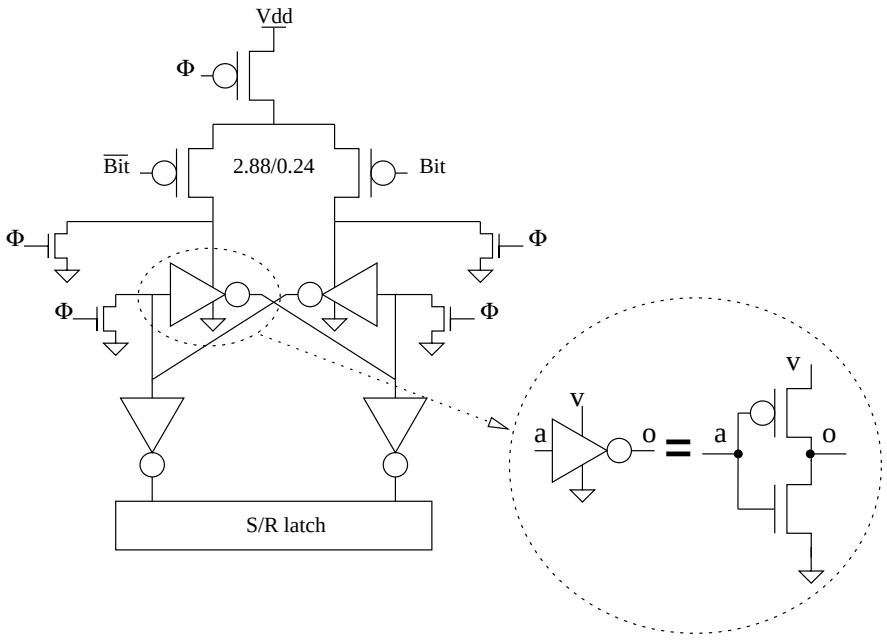

Figure 2.18: Example low-swing interconnect receiver, sized for a 0.18μm technology[54]

show an example low-swing interconnect system. The wires are precharged to Vlow, a voltage near ground. The drivers conditionally pull the wire to Gnd based on the data. The modified StrongARM latch [57] at the far end amplifies the low-swing differential input voltage to a full-swing signal.

Low-swing signaling can, however, require an extra supply and precise clocking of the receiver sense amplifiers. Additionally, the small swings can be susceptible to noise, but this can be mitigated by careful circuit design and layout [39][40][54]. The low-swing drivers and receivers can be merged with the routing logic in a “multiplexed address line” design.

Since a large portion of the access delay can be in the transport, we can increase the cycle rate of the memory by pipelining in the transport. Many microprocessors already allocate multiple cycles for a cache or even a register file access due to the interconnect delay. By pipelining in the transport we minimize the necessary latching elements by placing them before the block decoder.

2.5 Memory Systems

In the previous sections, we have reviewed the basic architecture of a partitioned memory array and a number of contemporary SRAM circuit techniques for achieving high-performance and low-power. These highly optimized SRAMs are rarely used by themselves, but rather aggregated together to form more complex memory systems. On modern processors and ASICs, the on-die memory system occupies a large percentage of the die area and has a large impact on performance and power dissipation.[3] For current system-on-a-chip ASICs, the memory occupies on average over half of the die area [58][59]. Similarly, memory occupies a large portion of the die on modern general purpose CPUs. For example, on a recently reported Intel Itanium 2 processor, the caches account for over 73% of the total device count and occupy over 50% of the die area [60].

Digital systems use memories for a variety of functions, but they typically fall into one of three categories: scratchpads, caches, and FIFO buffers. Scratchpads are simple, fast, software-managed, local memories mapped directly into the address space for storing frequently used data and instructions [61]. For fast, deterministic access times, scratchpads are typically small and located close to the computation. On the other hand, caches facilitate fast access to much larger memory spaces for access patterns that exhibit temporal or spatial locality by remapping locations of a larger memory space into a smaller, faster cache memory. Caches improve performance and efficiency of the memory system by hiding the latency of large, slow storage or by filtering requests to bandwidth limited resources such as off-chip DRAM. Some access pattern, however, do not exhibit locality, but rather streaming behavior [62][63][64]. Theses access patterns perform most efficiently with a FIFO stream buffer memories [65][66][67][68]. FIFO structures can also be used to provide elasticity between blocks with variable or bursty computation rates.

Even within each memory type there can be a multiplicity of different functionality. For example, while a cache for a large microprocessor may store coherence protocol information, replacement policy information, speculative state, or dirty information for each

[3] While the on-die memory itself may not necessarily be the dominant source of power dissipation, it can have a large impact on the number of off-chip memory requests issued. This off-chip I/O can be quite expensive in both power and performance. Additionally, the leakage power from large memory structures can be significant.

line. These extra meta-data bits may require special handling such as read-modify-write functionality. However, a cache for a small micro-controller may simply store a valid bit, and not require any additional functionality. Further, the large microprocessor's caches are likely to be multi-way set associative and organized in a hierarchy, while the micro-controller may only have a single level of relatively small caches. While these memory structures can have widely varying functionality, on closer examination, all of them are quite similar in implementation. Each of them can be decomposed into a number of RAM blocks, interconnection between the blocks and the computation, and interface logic between the computation and the interconnect.

Processors and ASICs often include examples of all three types of memories in various sizes and forms on the same die. On a processor, the are many caching structures (*e.g.* main cache hierarchy, branch target buffer, TLB), FIFOs (*e.g.* instruction queue, reorder buffer, reservation stations, DRAM write buffer), and scratchpad memories (*e.g.* register file). Even on more application specific chips such as multimedia processors and ASICs, there can be a mixture of these memory types [69][70][71][58]. The reason for this variety of memory structures on-die is that different parts of the design have different memory requirements, and each memory type is most efficient for a certain type of access patterns. On a larger scope, the memory systems of ASICs targeted at different applications vary widely, because they are optimized for different applications with different memory access patterns.

Chapter 3

Reconfigurable Memory Architecture

A conventional hardwired memory system is typically implemented for best average-case performance and efficiency, which is acceptable for an ASIC with a very narrow application range, but for a general purpose design, a hardwired memory system that cannot adapt to the varying memory access characteristics across a range of applications can be a substantial hindrance. Because the memory system is often the bottleneck for applications [17], a sub-optimal memory system can significantly degrade the performance and efficiency of the design [18][72][73][74]. If we had a configurable memory system, each application or application segment could have a memory system tailored to its specific needs and thus run faster and more efficiently [75][76].

A number of researchers have sought to build memory systems that could adapt to the application characteristics. The concept of caching is a way to dynamically reconfigure the memory to suit the application based on its past memory access behavior. Researchers have further enhanced cache performance and efficiency by altering the cache structure itself to meet the application needs. These enhancements have included reconfiguring the line size [72][77], the set associativity [78][18][72][79], and shutting off unneeded lines or banks [80][81][82]. Some proposed caches have a direct access, scratchpad-like mode that eschews the tag check to save power [83]. A few DSP chips can even redistribute memory resources between instruction cache, data cache, and scratchpad memory, on a cache way granularity [84][85]. Some researchers have gone further by adding the ability to alter the balance between compute and memory by reconfiguring the caches to become computation

[86].

Reconfigurable memory of a different sort can be found on commercial FPGAs and coarse-grain reconfigurable computing fabrics. To provide the local memory that applications need for buffering and local data storage, designers have included block RAMs distributed throughout the computing fabric which have lower overheads than using the configurable logic blocks (CLBs) as memory [6][5][87][14][88]. These block RAMs can reconfigure their access width, many can become FIFOs, and some can be configured as CAMs or logic blocks [89][90]. These designs provide relatively bare memory modules and use the surrounding configurable logic fabric to create any higher-level functionality needed by the system. However, the high overhead associated with the configurable logic degrades the performance of more complex memory functions.

The extant work on reconfigurable memory has focused on the architectural aspects of reconfiguring the memory, but has largely ignored the circuit-level effects of adding reconfiguration to a cutting-edge memory design. As an example, the FPGA block RAMs specifications are fairly inefficient and slow when compared to current, cutting-edge, embedded SRAM designs [5][91]. In this work, we take a different approach to the problem by looking at reconfigurable memory design from a bottom-up, *tabula rasa* perspective, starting with the circuit design of the memories themselves, and then examining where and what architecturally useful configurability can be added for low overhead. To provide a realistic evaluation, we apply modern, full-custom, circuit techniques to both the SRAM and the peripheral logic, which allows us to achieve performance and power on par with contemporary embedded SRAMs. To guide our development of the additional logic, we target three different memory configurations: caches, FIFOs, and scratchpads, but the resultant reconfigurable memory design is not limited to these configurations.

3.1 Reconfigurable Memory Architecture

Our reconfigurable memory systems consists of three sections: the memory, the interconnect, and the interface logic. Figure 3.1 shows the block diagram of the architecture. For our design, we leverage the natural partitioning of large SRAMs into smaller blocks, by adding reconfiguration on these partitioning boundaries. By choosing the reconfiguration

grain size based on circuit- and VLSI-level concerns, we avoid over or under partitioning the array and minimize the reconfigurability overhead costs.

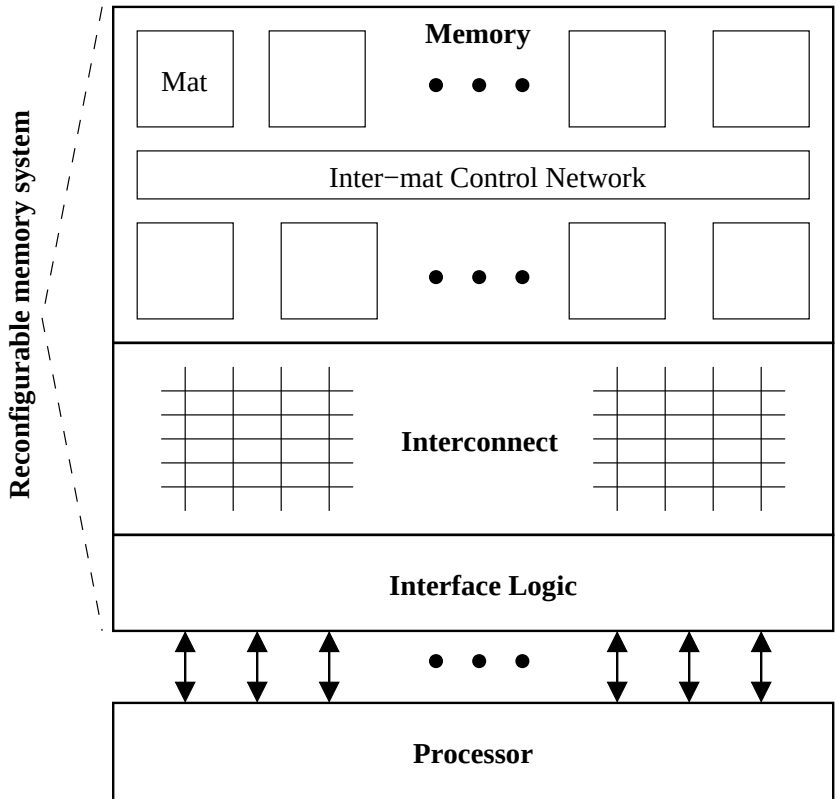

Figure 3.1: Basic reconfigurable memory system architecture

The memory section consists of a homogeneous array of memory blocks called *mats*. Each mat can be configured to be a portion of a cache, a FIFO, or scratchpad memory. We choose the mat size based on the optimal energy-delay SRAM block size and the necessary architectural flexibility. A small network called the inter-mat control network runs between the mats allowing them to pass a few bits of control information to each other. Mats can be aggregated together to form larger complex memories such as caches, FIFOs, or scratchpad memories. Figure 3.2 shows the block diagram of an example reconfigurable memory system with 16 mats arranged as an 8 x 2 array. Figures 3.3 and 3.4 show this memory system configured for caching and for stream processing respectively.

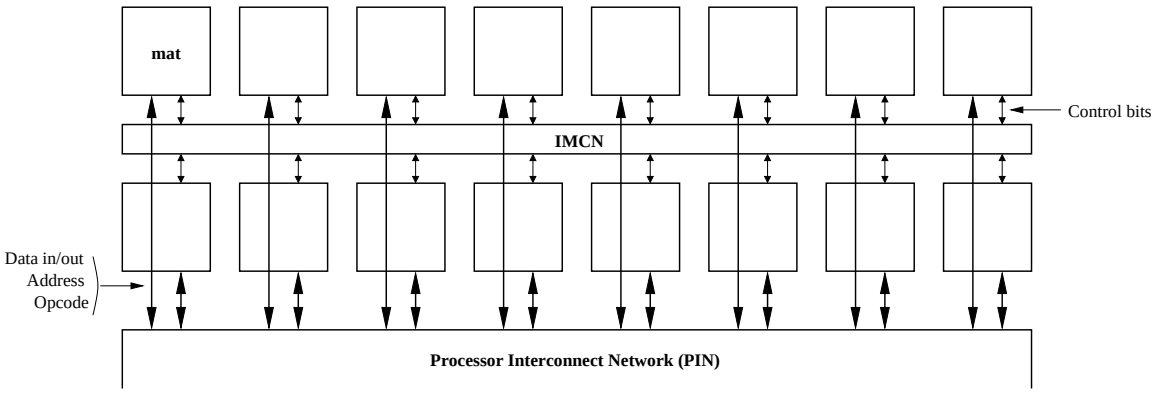

Figure 3.2: Memory system block diagram - 16 mats in an 8 x 2 array

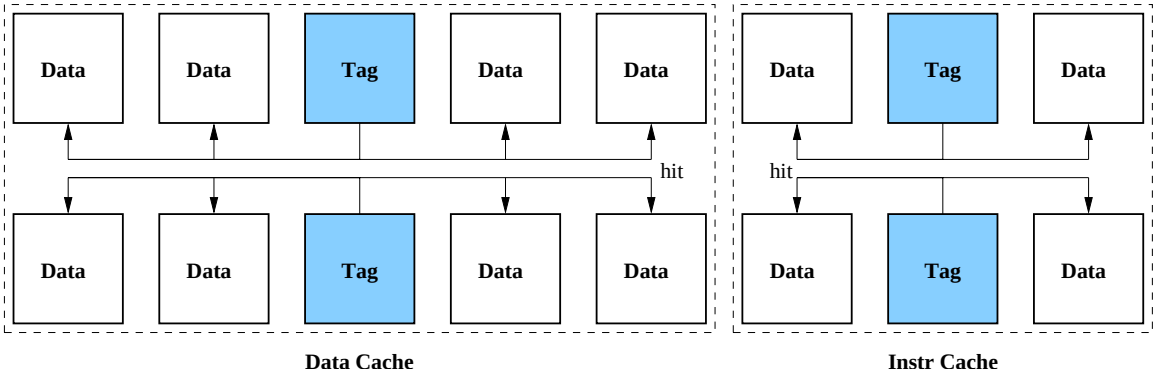

Figure 3.3: Caching configuration with 2-way data and instruction caches

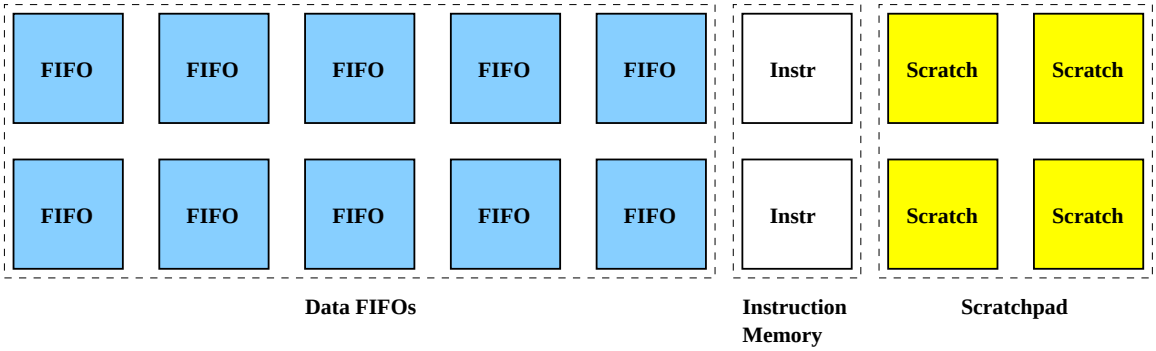

Figure 3.4: Streaming configuration with data FIFOs, instruction memory, and scratchpad

Because many memory structures are internally very similar, we can use this generalized memory mat to form the core of many types of memories. However, memory structures do differ significantly in the way that they connect to the computation engine. For example, a cache requires requests to two (or more) memory blocks, the tag array and the data array. A FIFO, however, only requires a single memory request per access. To allow maximum reconfigurability, we use a dynamically routed crossbar between the memories and the computation. Each mat has its own independent port into the interconnection network. The interconnect is actually two uni-directional interconnection networks: one for memory requests from the computation to the mats, and one for the reply from the mats to the computation engine. The computation engine launches requests into the interconnect, which are then dynamically routed to the addressed mats. The interconnect allows for multi-casting from a single request port to multiple mats. The mat replies are always routed back to the computation engine port that issued the request.

Memory structures can be quite varied in the way that they exist in the address space. A scratchpad is typically mapped directly into the address space. However, a cache has no mapping into the address space, but rather is a surrogate for the main memory, and must maintain the illusion that accessing the cache location is logically the same as accessing the main memory location. A FIFO may be addressed as a unit, rather than on a per word basis, with simple push and pop commands. The interface logic between the processor and interconnect translates the address from the processor to a hardware location. This logic is similar to the address centrifuge of the Cray T3E [92]. Chapter 4 provides more details on the interconnect and interface logic.

3.2 Memory mat

A memory mat is a flexible memory block which can emulate a wide variety of memory structures. The mat has a fixed access width, corresponding to the data word width of the computation engine. While having variable access width would facilitate applications that operate on smaller data widths, it decreases the efficiency of the memory [30] and complicates the design of the mat, interconnect, and interface logic. We can however aggregate multiple mats for wider access widths.

One of the key features that distinguishes many of the memory structures we wish to emulate from basic RAMs is that they store status bits for each word of data. For example, a cache tag may store a valid bit, the LRU status, the coherence protocol state, or the speculation state for each cache line. A FIFO may store a full/empty bit for each data word to indicate if the word holds valid data. In our reconfigurable memory, we support these status bits by adding a generalized status bits called *meta-data* bits to each data word. These bits hold data about the data, hence the name "meta-data." The meta-data bits are read and written along with the main data on all accesses. Additionally, there are special operations that only operate on the meta-data.

3.3 Operations

A mat can perform three basic operations: *read*, *write*, and a special meta-data operation called a *gang* operation. A read reads the requested word, data and meta-data, from the mat memory array and sends it to the mat data output. A write writes the accessed word, data and meta-data, with the word presented to the mat data input. A gang operation operates on all of the meta-data bits in the mat on a per-column basis in a single cycle.

A gang operation can set, clear, or leave alone (NOP) any combinations of meta-data columns. For example, if there were 4 meta-data bits, *md[3:0]*, with a single gang operation, we could set *md[3]*, clear *md[2:1]*, and NOP *md[0]*. Figure 3.5 illustrates this example gang operation.

There are two additional mat operations used for reading and writing the configuration state, *configuration read* and *configuration write*. The configuration state is memory mapped into a special address space accessible only via the configuration read and write operations. The ability to read the configuration state is primarily a testability feature and not strictly needed.

3.4 Operation Modifiers

There are four operation modifiers than can be applied to the three basic operations per Table 3.1. An "X" indicates that the modifier can be applied to the operation. The modifiers

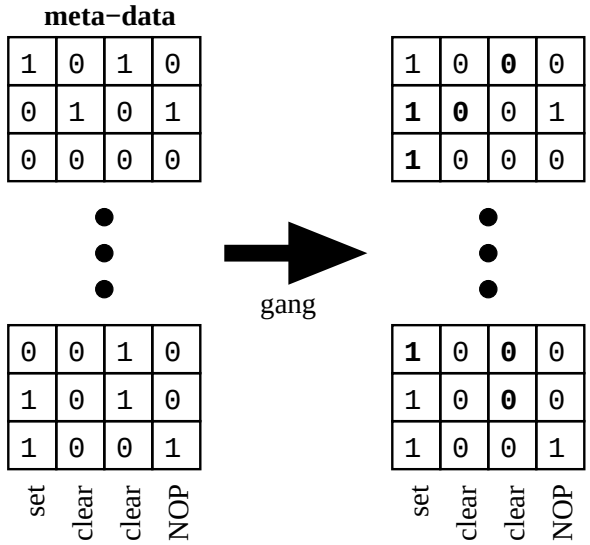

Figure 3.5: Example gang operation - set md[3], clear md[2:1], NOP md[0]

Table 3.1: Mat operation modifier applicability

Instruction	Cmp	Ptr	RMW	Cond
Read	X	X	X	X
Write		X		X
Gang				X

are compare (cmp), pointer (ptr), read-modify-write (RMW), and conditional (cond). All modifiers are orthogonal.

3.4.1 Comparisons

One of the most common logic operations following a memory read is a compare. For example, the vast majority of accesses to a cache tag memory are reads followed by a compare to determine if there is a cache hit. Some configurations may only want to perform a comparison on a portion of the stored word. For example, in a cache tag that stores both LRU and valid information in the meta-data, a tag check will want to compare the outgoing address to the stored tag and ensure that the valid bit is set by checking that the valid bit

is set. But the check will not care about the state of the LRU information. So we want compare the main data and the valid bit, but wildcard the LRU bits.

The memory mat supports a maskable compare modifier, that follows up a read operation with a comparison. Unlike a normal read, a compare requires data input for the value to compare the stored value against. Additionally, for the maskable comparison, we need a short mask field to determine which fields are to be compared. The mask field is one bit longer than the amount of meta-data to allow us to mask out any combination of the meta-data bits and the main data as a chunk. Comparator hardware is relatively small, so embedding a comparator in each mat only has a modest area overhead.

3.4.2 Pointer Operations

Many memory structures have a level of indirection in the address path. A simple example of this is a FIFO, in which an access is either a pop of the head or a push to the tail. A request does not name a specific memory address to access, but instead a pointer, the head or tail, to access. Internally, the FIFO maps the head and tail pointer to an actual memory location. A more complex example of address indirection occurs in a cache, where the accessed word is selected based on matching the memory access tag field. Our memory system requires use of two (or more) mats to support this type of complex address indirection, one to store the tags and another for the data.

We support one level of simple address indirection via pointer operations. Any read or write can be a pointer access that names a pointer number to access, rather than a memory address. A special block in the address path called the *pointer logic* translates this pointer number into a memory address. Each pointer has an associated stride value that the pointer logic uses to update the pointer value after an access. The pointer value updates are optional and can be increments or decrements. The pointer and stride values are read or written using configuration reads and writes, described later.

3.4.3 Read Modify Writes

A read-modify-write (RMW) operation is an atomic operation that reads out a word of data, performs a logic function on the meta-data, then writes the modified meta-data back into

the previous accessed word. The logic function may take other inputs besides just the read out meta-data. A write-modify-write operation is also possible, but the usefulness of such an instruction is somewhat suspect.

An example use of this operation would be in a cache tag that stores LRU information for each line. In a mat configured as the tag, the LRU bit(s) would be stored in the meta-data. When the tag is accessed, we calculate the new LRU value based on the previous LRU value, the local hit/miss result, and the global hit/miss result. By using a RMW operation, we can perform this very simple logic and write the meta-data back all within the memory. While the operation implementation may be pipelined internally, we maintain the appearance of atomicity to all requesters. So if a RMW is immediately followed by a read of the same word, the second read returns the updated meta-data value generated in the modify logic.

3.4.4 Conditional Operations

Many memory structures have operations whose execution is contingent on an internal or external condition. An example of a conditional operation based on an external condition would be the data write of a cache. In the data memory, the write should only occur if there is a hit in the tag. The hit signal must be passed from the tag memory to the data memory to tell it whether or not to execute the data write.

A conditional operation based on an internal condition is contingent on a pattern match against the meta-data of the word. An example of this would be a FIFO push, a write into tail pointer location. The write is conditional on the FIFO having an empty slot (*i.e.* the FIFO is not full). The write is therefore contingent on the meta-data bit storing the full/empty information being 0 indicating an empty location. An internal conditional operation specifies the condition similar to specifying the mask for maskable compares.

Any operation can be a conditional operation. A normal memory read does not alter the state of the data word, and thus a conditional read does not seem strictly necessary, but when aggregating multiple mats to form larger memory structures, a conditional read can be useful. For any operation than can change the state of the word (*i.e.* a write or a read-modify-write), a conditional operation is needed. A conditional gang operation requires

Table 3.2: Conditional gang clear truth table

md[1]	md[0]	md[0]'
0	0	0
0	1	1
1	0	0
1	1	0

additional support as described next.

3.4.5 Conditional Gang Operations

A conditional gang operation gang sets or clears a meta-data bit based on the value of another meta-data bit. Figure 3.6 shows a conditional gang clear of *md[0]* based on *md[1]*. If *md[1]* is 1, we clear *md[0]*, otherwise *md[0]* is left alone. Table 3.2 shows the truth table for this operation.

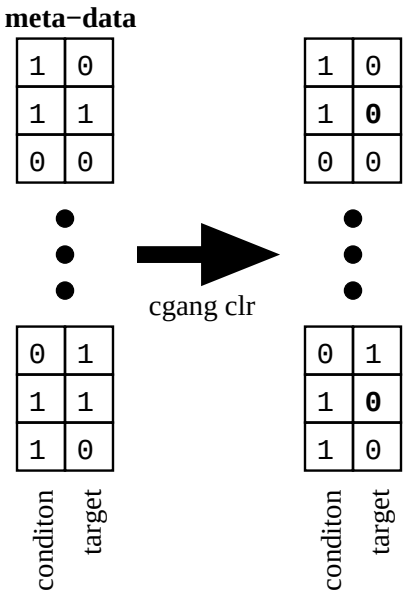

Figure 3.6: Example conditional gang clear operation - clear *md[0]* if *md[1]* == 1

Table 3.3: Mat I/Os

Signal	Bits	Description
opcode	o	pre-decoded one-hot control signals
address	a	memory address or pointer number
mask	m+1	bit mask for compares and gang ops
mdata_in	m	meta-data input
data_in	d	data input
mdata_out	m	meta-data output
data_out	d	data output
ext_in	e	external control input
ext_out	e	external control output
match_out	1	compare result
valid_out	1	valid status of output data
complete	1	general completion signal

Systems that allow speculative memory operations such as the Hydra multiprocessor [93] can make extensive use of conditional gang operations. Each processor in Hydra runs a different thread, all but one of which is speculative. Hydra caches have a number of special status bits that they use to keep track of the speculative state of the cache lines. The Modified bit keeps track of whether a line has been speculatively written by the local processor, set to 1 if it has. This bit would be stored in the meta-data of a mat acting as the cache tag in a Hydra implementation on our reconfigurable memory system. If a processor speculates incorrectly (*e.g.* speculatively reads a word that is later written by a less speculative thread), then it must perform a backup and invalidate all lines in the cache that it has speculatively written. If the Modified bit is set, then the processor has incorrectly speculatively written the line and must clear the Valid bit. This requires a conditional gang clear of the Valid bits based on the Modified bits.

3.5 Mat Interface

In order to support the variety of operations beyond simple reads and writes, the mat interface is more complex than a basic memory. Table 3.3 lists the interface signals. The *opcode*

signal is *o* bits wide and indicates what operation the mat should perform. This takes the place of the usual read and write enable signals. The mat address is *a* bits wide, meaning the mat stores 2^a words. In pointer and gang operations that do not need to provide an address, we embed other necessary operation specifiers in the address. Pointer operations embed the pointer number, update enable, and update add/subtract specifier. Gang operations embed the *gang_data* field, which is used in conjunction with *mask* to determine which columns to set, clear, or NOP. Compares also use the *mask* field to determine which bits to use in the compare. The *mask* field is one bit wider than the meta-data, because compares can mask out the main data word as a chunk.

Each word has *d* bits of main data and *m* bits of meta-data. The interface to the IMCN used for communicating control information to/from other mats is via the *e* bits wide *ext_in* and *ext_out* signals. The mat exports control data on *ext_out* and receives it on *ext_in*. The *match_out* signal is the result of the comparison. If the operation is not a compare, the signal remains low. The *valid* signal indicates to the interconnect that the mat has valid data to output. The *complete* signal is a general purpose completion signal to tell the computation that the memory request has executed successfully.

There are many ways to implement this reconfigurable memory mat architecture. The next section presents a full-custom implementation that was used in the prototype testchip discussed in Chapter 5.

3.6 Micro-architecture and Implementation

This section examines a full custom circuit design implementation of the reconfigurable memory mat architecture that emphasizes tight integration with the SRAM core, low latency, fast cycle time, and low overhead. Chapter 5 discusses additional details and experimental results from the prototype testchip using this design. The SRAM design uses the techniques discussed in Chapter 2 for high performance and efficiency.

To a basic SRAM array, we add meta-data and peripheral logic blocks to create a flexible memory mat. The overarching goal for these additional logic blocks is to have small performance, power, and area overheads, and yet be general and flexible enough to meet the needs of many memory configurations. We generalize the logic blocks and functions

as much as is feasible to increase hardware sharing across different configurations and to increase flexibility.

For this peripheral logic, we primarily used hardwired logic, rather than reconfigurable logic such as the look-up tables (LUTs) used in FPGAs, because we only need a few, relatively simple peripheral logic blocks. By using hardwired logic, we ensure that the logic will be small, fast, and efficient, at the cost of having some unused peripheral logic blocks in some configurations. This also reduces the amount of configuration state and reconfiguration time, because we don't have large LUT arrays to program.

The peripheral logic can be broken down into two basic blocks: address logic and datapath logic. The address logic sits between the address input and the memory array decoder and operates on the address bits. The datapath logic sits between the data I/O and the SRAM core and performs logic on the data read from or written to the memory array. Figure 3.7 shows a block diagram of a generic memory building block. Our design adds the necessary peripheral logic blocks for the target configurations, as shown in Figure 3.8.

Despite adding this peripheral logic for additional functionality, we wish to maintain high performance. Thus, we chose an aggressive cycle time goal of 10 fan-out-of-four inverter delays (FO4) based on the achievable access times for SRAM arrays in the optimal energy-delay size range. This short clock tick stresses the circuit design of the memory cores as well as the peripheral circuits. Due to this aggressive cycle time, the mat access is pipelined, and the total delay through the mat is 2 cycles or 20 FO4. The first half-cycle is spent in the pre-access logic: pointer logic or write buffer. The next full cycle is spent in the SRAM access. The last half-cycle is spent in the post-access logic: control logic and the comparator. The mat is fully pipelined, accepting a new request every 10 FO4 cycle.

In comparison to our memory system's 10 FO4 clock cycle, state-of-the-art microprocessors generally run at 15-20 FO4 [94], a typical synthesized custom ASIC at 40 FO4, and FPGAs at around 100 FO4.[1] While a few compute blocks [95] and caches [96][97] operating at approximately 10 FO4 have been demonstrated in commercial microprocessors, these units were usually internally "double pumped," operating at twice the main clock rate. One notable exception is the IBM/Toshiba/Sony Cell processor which does clock at

[1] While the theoretical maximum clock frequency of FPGAs is usually around 50 FO4, configured FPGAs typically run at around 100 FO4.

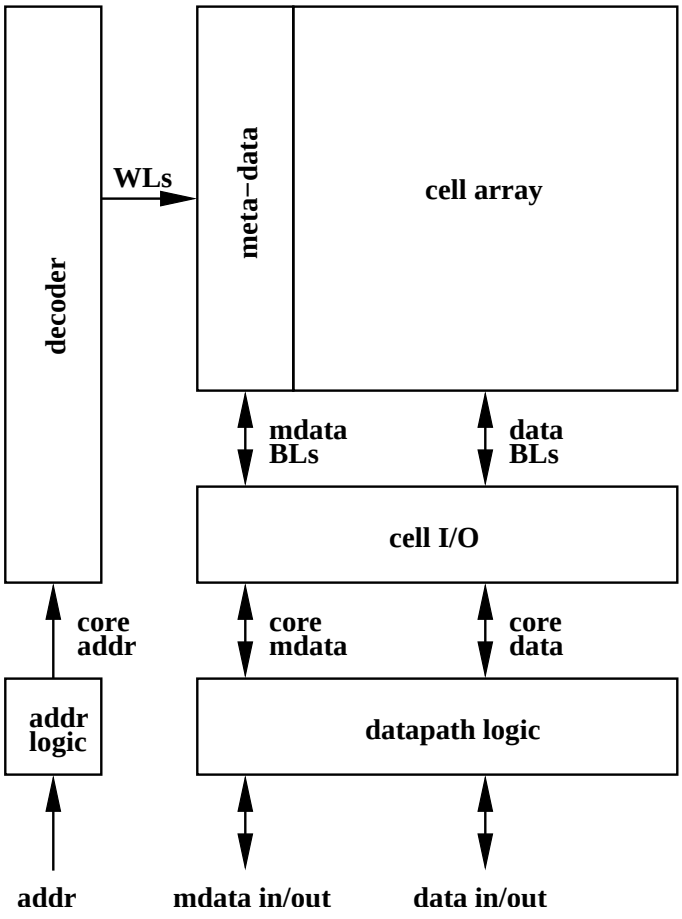

Figure 3.7: Generic memory structure

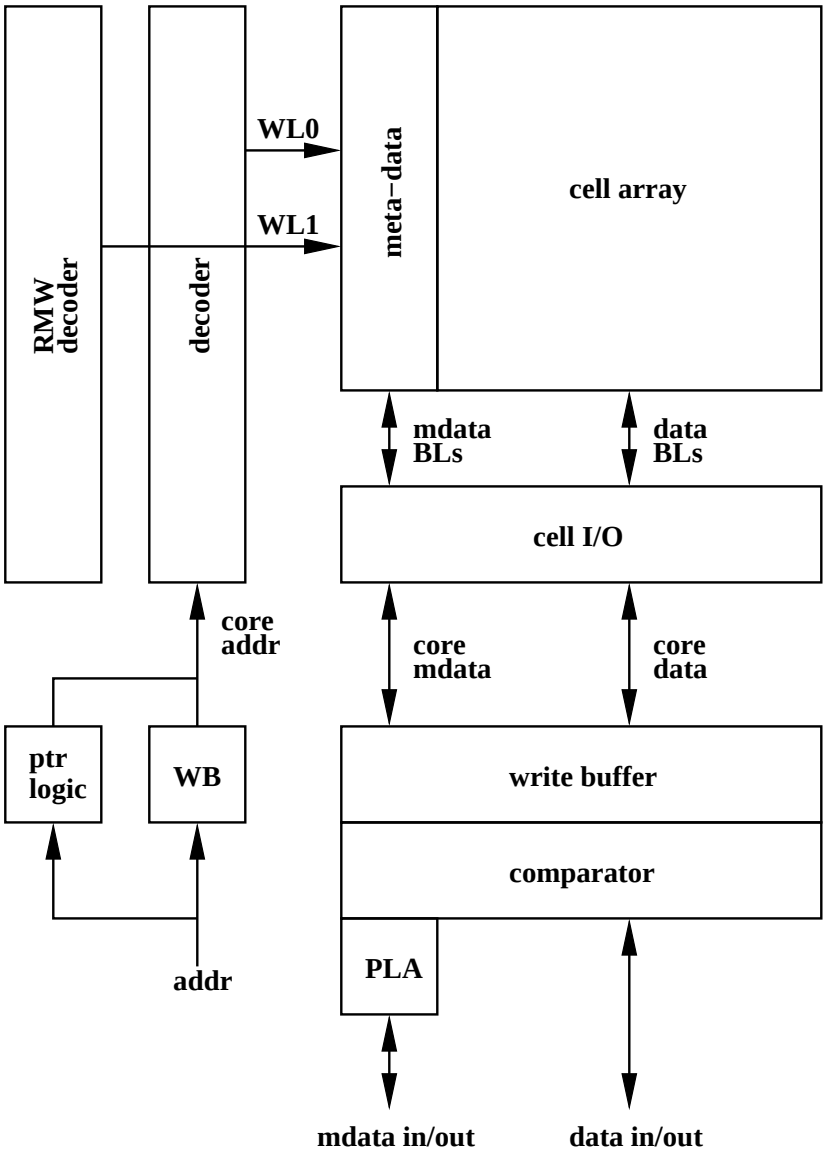

Figure 3.8: Mat block diagram showing meta-data with support logic (RMW decoder and PLA) and peripheral logic blocks (pointer logic, write buffer, and comparator)

Table 3.4: Clock cycle comparison for virtual multi-porting

	Clock cycle (FO4)	Frequency in 0.18μm (MHz)	Accesses per cycle
Memory	10	1000	1
CPU	20	500	2
ASIC	40	250	4
FPGA	100	100	10

11 FO4 [98]. The local memory of the Cell processor is pipelined [99] to meet the fast clock cycle.

Running the memory system at a faster clock rate than the processor allows us to virtually multi-port the memory. During a single processor cycle time, we could access the memory multiple times, thus emulating the functionality of a memory that has multiple read/write ports. Even for a state-of-the-art microprocessor running at 20 FO4, we could make two memory accesses per cycle with our 10 FO4 memory system. For an ASIC or FPGA, we could make many more accesses per cycle, as detailed in Table 3.4.

By using a fast single-ported memory, rather than a true multi-ported memory, we can achieve significant savings in both area and power, especially in systems that demand a large number of ports [100]. In a virtual multi-ported memory, the accesses are staggered, because the accesses are not truly simultaneous. This is less of a factor in systems where the memory runs much faster than the processing units, since the memory accesses are only staggered by a small amount of time (*i.e.* one memory clock cycle) from the processing unit's perspective.

To meet our aggressive access and cycle time goals, we employ the highly optimized circuit techniques used in modern SRAMs for high performance and low power throughout the design. In this way, the peripheral logic can match the performance and power characteristics of the memory core. The next section focuses on the implementation of the meta-data bits, because they are the key feature that enables the configurable memory and required the most circuit innovation.

3.6.1 Meta-data

We tightly integrate the meta-data with the main data SRAM array to share as many resources (*e.g.*, decoder, replica control path) as possible for low overhead and high-performance. Thus, we implement the meta-data using additional storage cells in the main SRAM array (Figure 3.8). The meta-data bit cell, shown in Figure 3.9, is an explicitly two-ported cell to efficiently support RMW operations.

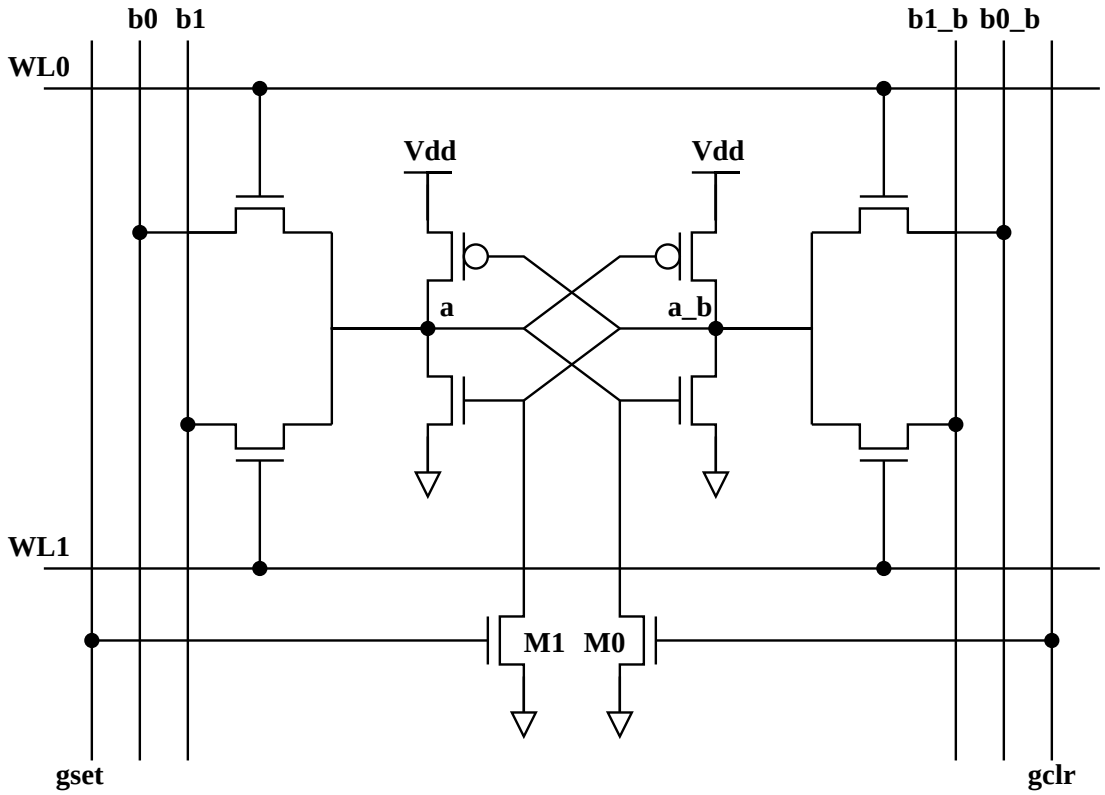

Figure 3.9: Meta-data bitcell

All normal accesses use port 0 (*WL0, b0, b0_b*). The main decoder drives *WL0* which goes to all meta-data cells and all single-ported normal data cells in a row. Port 1 is a special port used by read-modify-write operations. Note that the special wordline *WL1* goes only to the meta-data cells as shown in Figure 3.10. In our prototype implementation, the single-ported data cell has a free horizontal metal track, which allows the meta-data cell

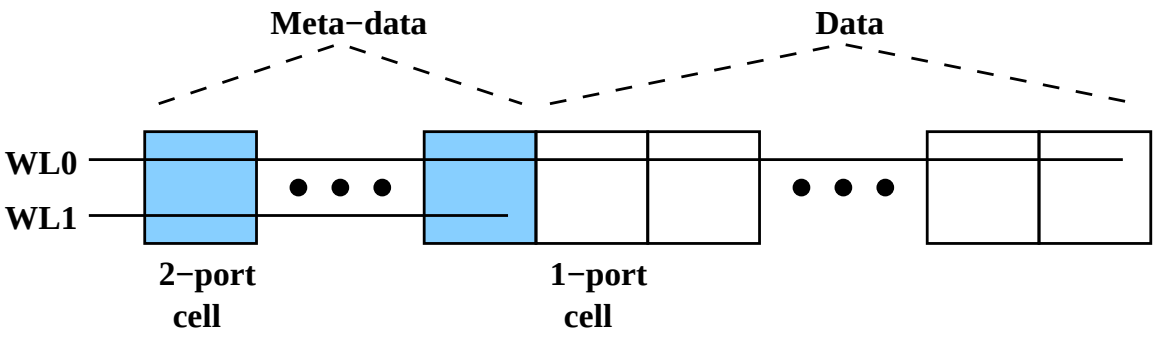

Figure 3.10: One mat cell row

to be laid out in the same cell pitch as the single-ported data cell, using the free metal track for *WL1* in the meta-data cells.

As was discussed earlier, we support a number of special operations on the meta-data. Gang operations allow us to operate on all bits of the meta-data on a per-column basis in a single cycle. Any meta-data column can be gang set, gang cleared, or left alone (NOP). To implement gang operations, we add two additional devices in the meta-data bitcell *M0* and *M1*. All cells in a column share the *gset* and *gclr* lines. Asserting *gset* sets all cells in a column to 1. Asserting *gclr* clears all cells in a column to 0. Figure 3.11 shows the gang I/O logic which generates the *gset* and *gclr* signals. *Gang_en* is simply the gang operation enable signal.[2]

The additional wordline and gang control lines can increase the size of the meta-data bitcell. In some cell designs there is a free horizontal metal track, in which case, the cell height can remain the same as the normal single-ported data cells. The cell may then become wider due to the addition of the gang control lines, but will not adversely affect the vertical cell pitch. If there is no free metal track the meta-data cell will be taller than the normal data cells, and so if we wish to have the meta-data and data cells in the same row, the row vertical pitch will have to increase. This decreases the area efficiency of the memory, because the normal data cells are now no longer their minimum size. To avoid

[2]The same functionality could also be achieved by grounding the Vdd of one side of the cell. This may be more efficient, because it does not require additional devices or control lines in the cell. However, it does require that the cell Vdd's and Gnd's run vertically. The cell used in the testchip used horizontal Vdd's and Gnd's, so this technique was not adopted. Some modern cell designs do run Vdd and Gnd horizontally, and thus could use this technique to implement gang operations.

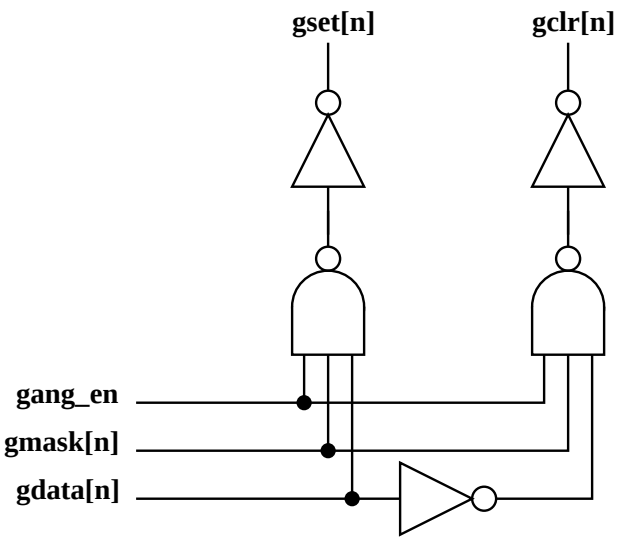

Figure 3.11: Gang operation I/O logic for one column

this problem, we can interleave the meta-data cells (see Figure 3.12) to keep the normal data cell pitch at a minimum.

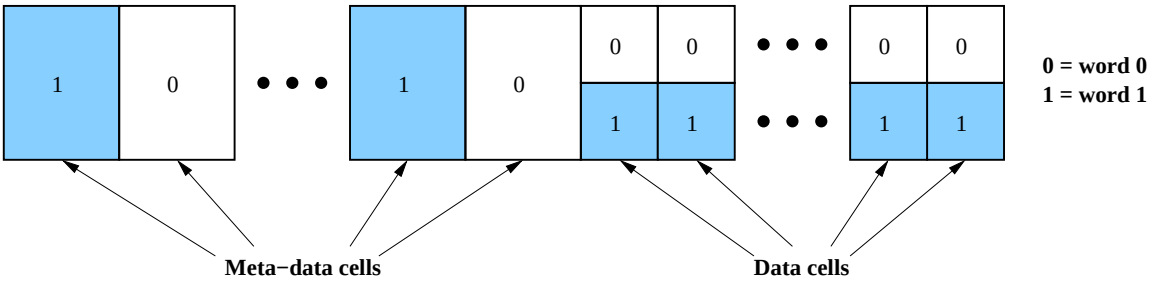

Figure 3.12: Two cell rows using interleaved meta-data bitcells

Conditional Operations

Conditional operations require additional logic in the meta-data cell. A conditional operation based on an internal condition is contingent on a match of the *mdata_in* with the stored value's meta-data. This could be performed as a pipelined 3-cycle operation, similar to a read-modify-write. However, because a conditional operation's final access may read or

write the main data, we would either have to stall the mat or add a second port to the main data word.

As an alternate implementation, we could add CAM-like compare logic to the meta-data bit cells and gate the main wordline based on the match result. Figure 3.13 shows the modified meta-data bit cell. The I/O logic would drive the *mdata_in* value to be matched against on the *K* and *K_b* lines masked by the *mask* input field. All meta-data bits masked out would drive a 0 on both *K* and *K_b* preventing a mismatch on that bit position. This implementation takes the match out of the critical path, because the match occurs in parallel with the main decode.

The wordline driver will be slightly slower due to the additional input of the match result. The additional match circuitry and signal lines increase the size of the meta-data bitcell. It is unlikely that the cell can be layed out in a normal single-ported cell pitch, and we would have to use an interleaved meta-data cell arrangement as shown in Figure 3.12. We would need to detect whether the operation successfully completed and return this result to the processor via the general purpose completion signal. This could be implemented using a wide OR of the wordlines. While this enables conditional reads and writes based on internal conditions, conditional gang operation are a special case discussed in the next section.

Conditional Gang Operations

Conditional gang operations also require special support circuits in the meta-data. Since most configurations that use conditional gang operations only need one bit conditionally cleared, we limit our conditional gang implementation to that function. We link two meta-data columns, with one column (*e.g. md[1]*) as the control column and another column (*e.g. md[0]*) as the target column. On a conditional gang clear, the target column is cleared if the meta-data bit in the condition column is 1. This is the same function previously described in Table 3.2.

A conditional gang clear can be implemented by linking two columns with a two NMOS pulldown stack (Figure 3.14). When *cgang* is asserted, if the cell in *md[1]* is 1, then the pulldown stack connects the a storage node of *md[0]* to Gnd, writing a 0 into the cell.

An alternate method of implementing conditional gang clear only uses a single NMOS

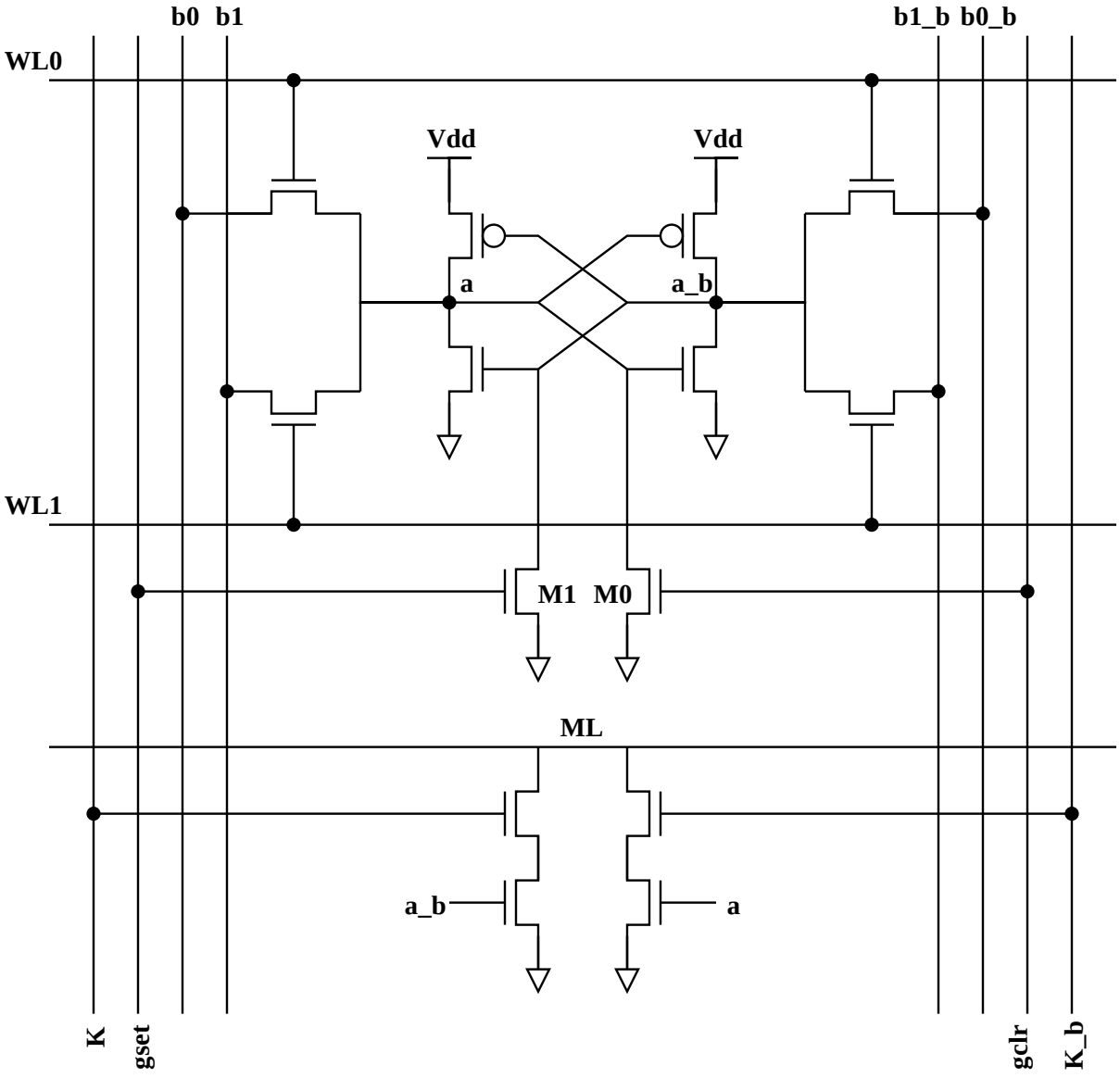

Figure 3.13: Meta-data bit cell with embedded match circuit

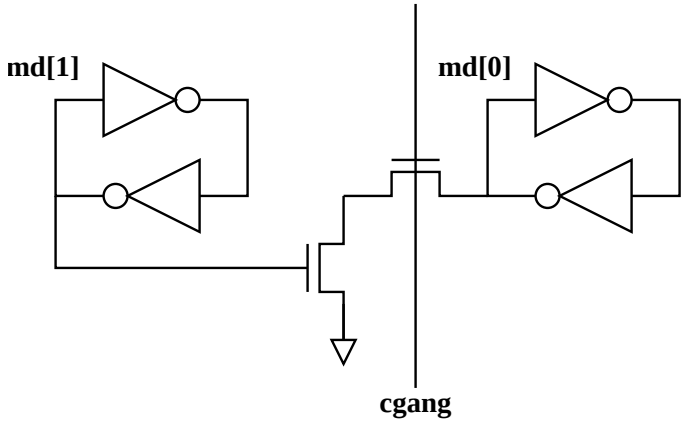

Figure 3.14: Conditional gang clear implementation using two transistors

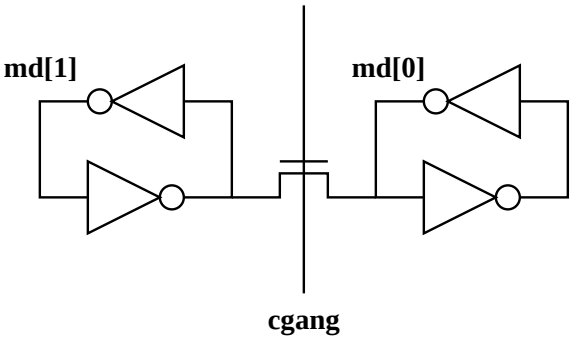

Figure 3.15: Modified conditional gang clear implementation using one transistor

Table 3.5: Modified conditional gang clear truth table

md[1]	md[0]	md[1]'	md[0]'
0	0	1	0
0	1	0	1
1	0	1	0
1	1	1	0

device to connect *md[1]_b* to *md[0]* with the device gate controlled by *cgang* (Figure 3.15). This performs the desired logic function on *md[0]*, but in the case where *md[0]* is 0, and *md[1]* is 0, a 1 will be written into *md[1]*. If *md[1]* will be gang cleared before further use (*e.g.*the Modified bit in Hydra is cleared on a speculative thread backup after the conditional gang clear of the Valid bits [93]) then this push-back from *md[0]* is a benign side-effect. Table 3.5 shows the truth table for the modified gang clear operation.

Because all other operations on the meta-data bits support either true or complement operations, having a uni-directional conditional gang operation still makes the meta-data bits logically complete. The sense of any of the bits could always be inverted without the loss of any functionality.

Read-Modify-Write Decoder

To support read-modify-write operations, we need an additional special decoder, called the RMW decoder, and a reconfigurable logic block to modify the meta-data bits. The latency of a read-modify-write operation is much longer than a single memory access, because it requires two memory accesses, a read and a write, plus the time for the modify logic. We pipeline read-modify-write operations to avoid adversely affecting the memory mat cycle time. Additionally, since a new operation can arrive each cycle, we use the second port of the meta-data bits for the RMW write so that it does not conflict with the incoming operation.

Thus, a read-modify-write operation is a pipelined, 3-cycle latency operation. In the first cycle, a standard read operation reads a word out of the array. In the next cycle, a reconfigurable logic block operates on the read-out meta-data bits. In the final cycle, the

output of the reconfigurable logic block is written back into the meta-data of the word accessed two cycles ago. From the requester's standpoint, this operation is atomic: any subsequent read of the word will retrieve the updated meta-data and no subsequent write will be over-written by the RMW writeback (*i.e.* no WAW hazard). The first cycle operation could also be a compare or even a write.

The RMW decoder remembers which word was accessed during the initial cycle of a RMW operation and drives the meta-data second port wordline during the write cycle. To minimize the number of forwarding paths, we want a simultaneous main read and RMW write to return the newest meta-data value. Thus the RMW decoder must fire the second port wordline before the main wordline activates. This need for fast RMW decoder operation eliminated the possibility of latching the read address and re-doing a full decode for the RMW write. Instead we chose to conditionally latch the wordlines using a crosscoupled inverter storage cell sized as an SRAM cell (Figure 3.16) on the RMW read. Figure 3.17 shows the timing diagram for the RMW decoder operation.

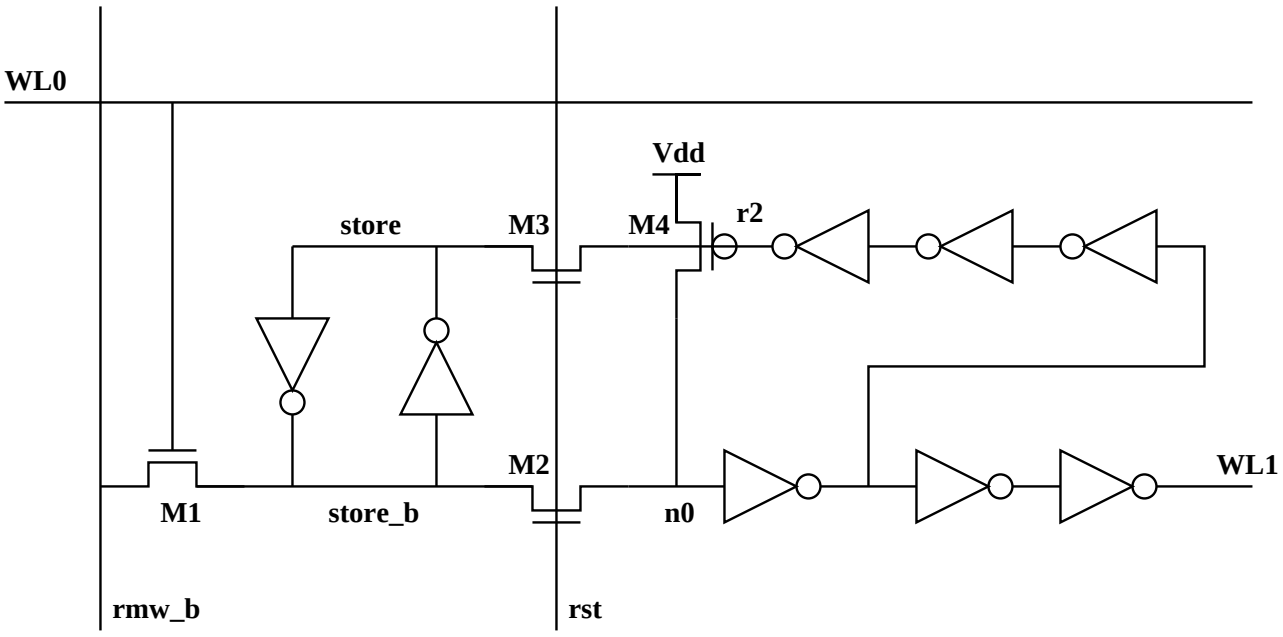

Figure 3.16: RMW decoder

During the RMW read cycle, the RMW I/O logic asserts *rmw_b* low. In the accessed

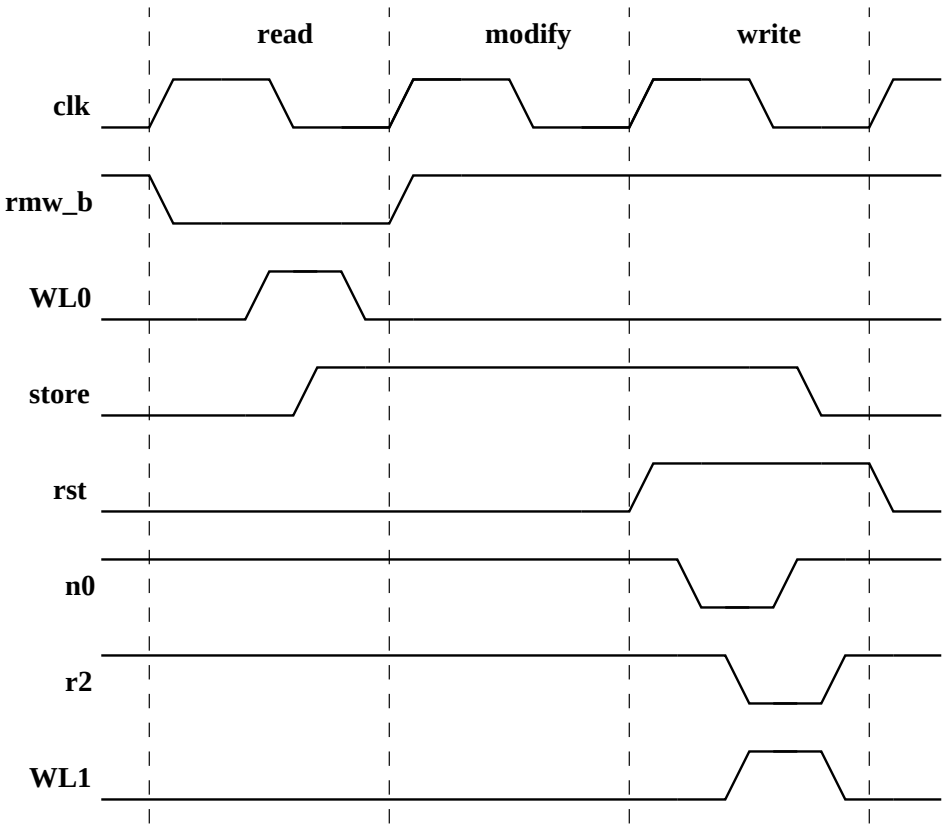

Figure 3.17: RMW decoder timing diagram

row, the main decoder pulses *WL0* which writes a 1 into the storage cell via *M1*. The *rst* signal stays low in this cycle. During the modify cycle, *rmw_b* is held high, and *rst* stays low. In the write cycle, *rst* rises which brings *n0* low through *M2* in the row that was accessed in the read cycle. *N0* going low asserts *WL1* and activates the self-reset delay chain. The delay chain drives *r2* low, turning on *M4*. *M4* then resets *n0* and *WL1*. As *r2* goes low *M3* clears the storage cell. The decoder can now accept another RMW operation. In rows accessed in the read cycle, the storage cell undergoes a "read" when *rst* asserts. *M3* must be carefully sized to avoid excessive power dissipation.

The *rst* signal is not just a delayed version of the *rmw_b* signal, because a write to the same word during the modify cycle aborts the writeback of the modified data. This condition must be checked in the *rst* drive logic. Because the writeback occurs early in the cycle, a write to the same word during the writeback cycle will overwrite the modified data. Because the mats are fully pipelined, we can have three RMW operations in-flight at any given time. Thus, we need three storage cells per row (Figure 3.18), and there are three sets of *rmw_b* and *rst* signals, *rmw_b[2:0]* and *rst[2:0]*. Which cell and signal pair is used rotates on a per cycle basis. On system reset, all RMW decoder storage cells are reset to 0 via a reset NMOS device (not shown).

Reconfigurable PLA

The other support block for RMW operations besides the RMW decoder is the reconfigurable logic block that performs the modify logic. There are many ways to design a reconfigurable logic block, but we chose to implement this block using a reconfigurable PLA (Figure 3.19), because PLAs are dense array structures suitable for small logic functions. The reconfigurable PLA is a NOR-NOR PLA (Figure 3.20), with the first NOR plane implemented as a ternary CAM, and the second NOR plane implemented as an SRAM with a special logic line. Because the PLA is basically a CAM and an SRAM, we can apply the fast, low-power SRAM circuit techniques seen in Chapter 2 to the PLA implementation.

The PLA can perform three operations: configuration read, configuration write, and logic. Configuration reads and writes allow requests to read and write the CAM and SRAM storage locations to change the programmed logic function. The logic operation performs the programmed logic function on the incoming meta-data bits. During a logic operation,

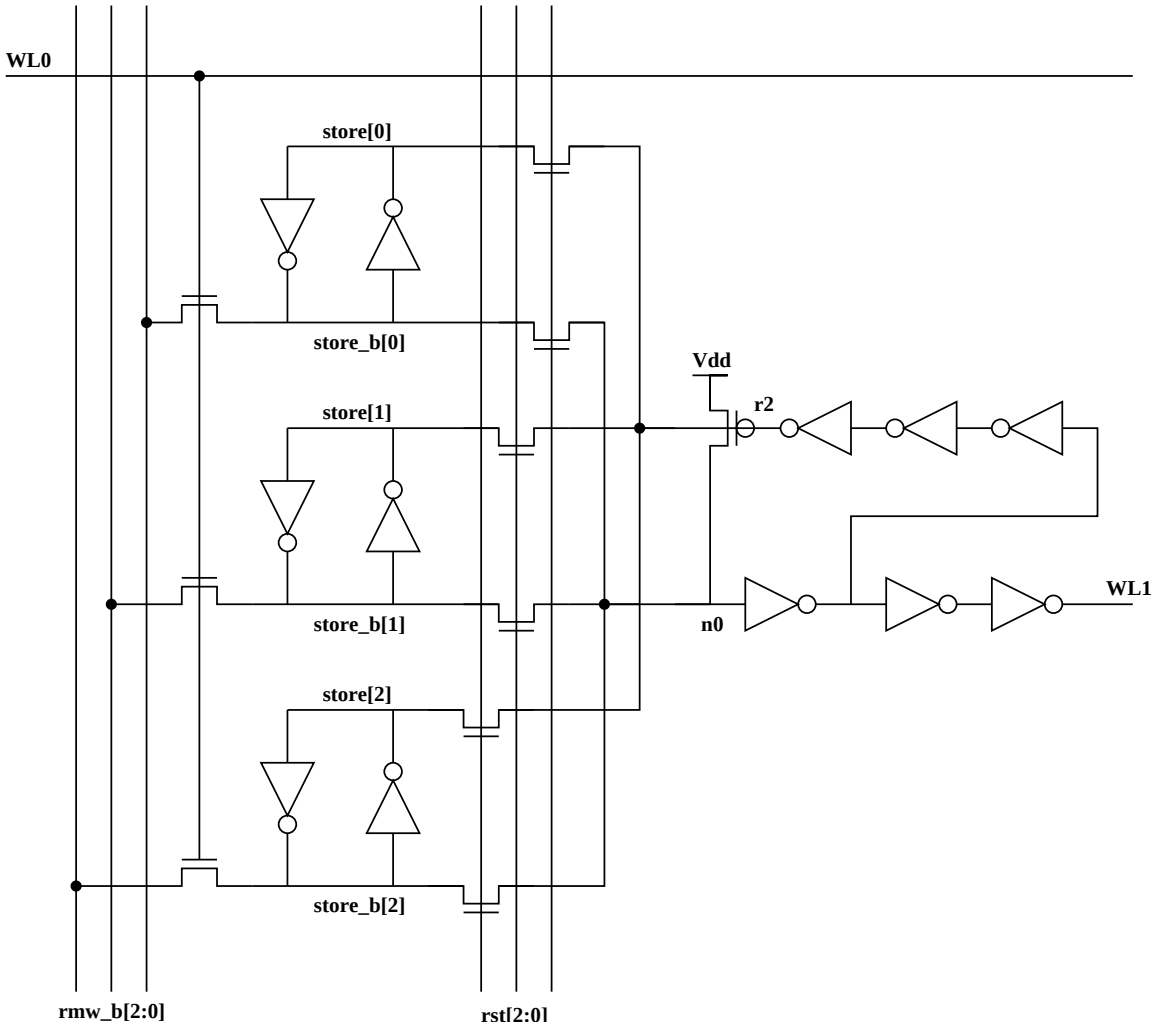

Figure 3.18: Pipelined RMW decoder

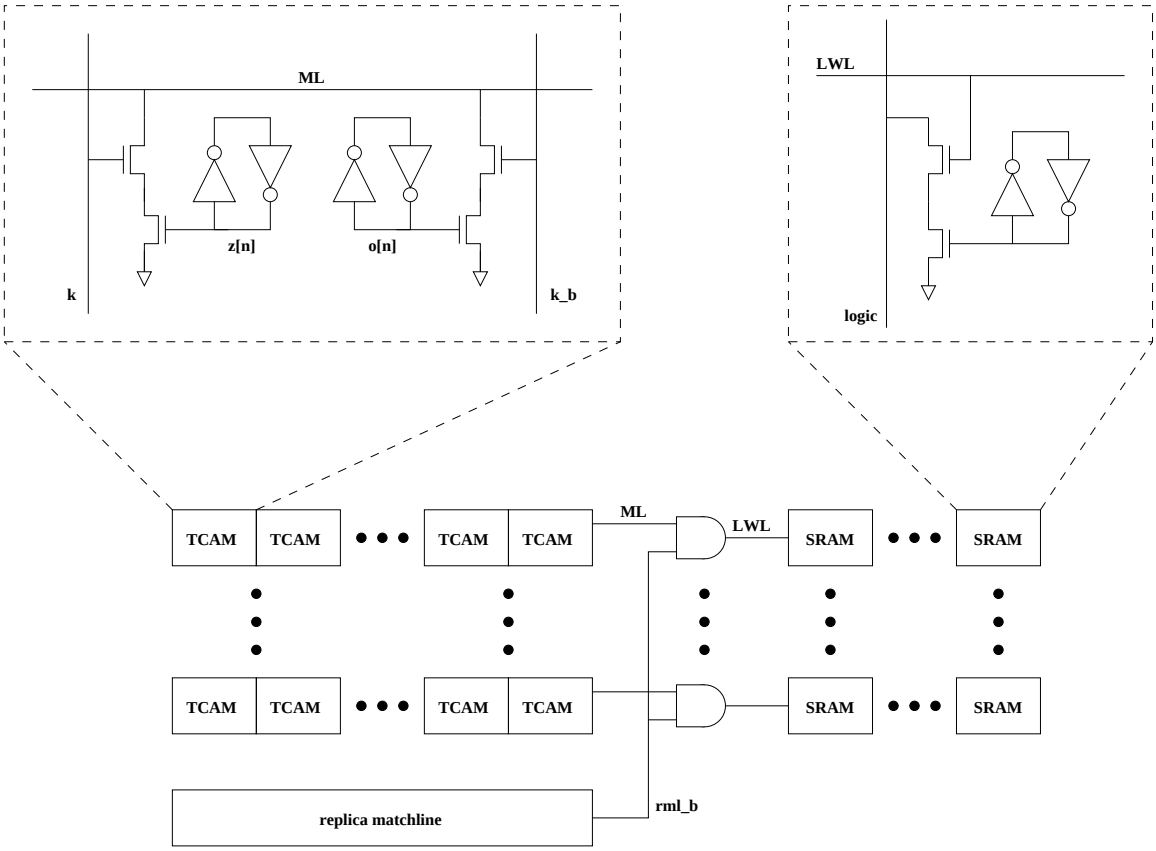

Figure 3.19: PLA block diagram

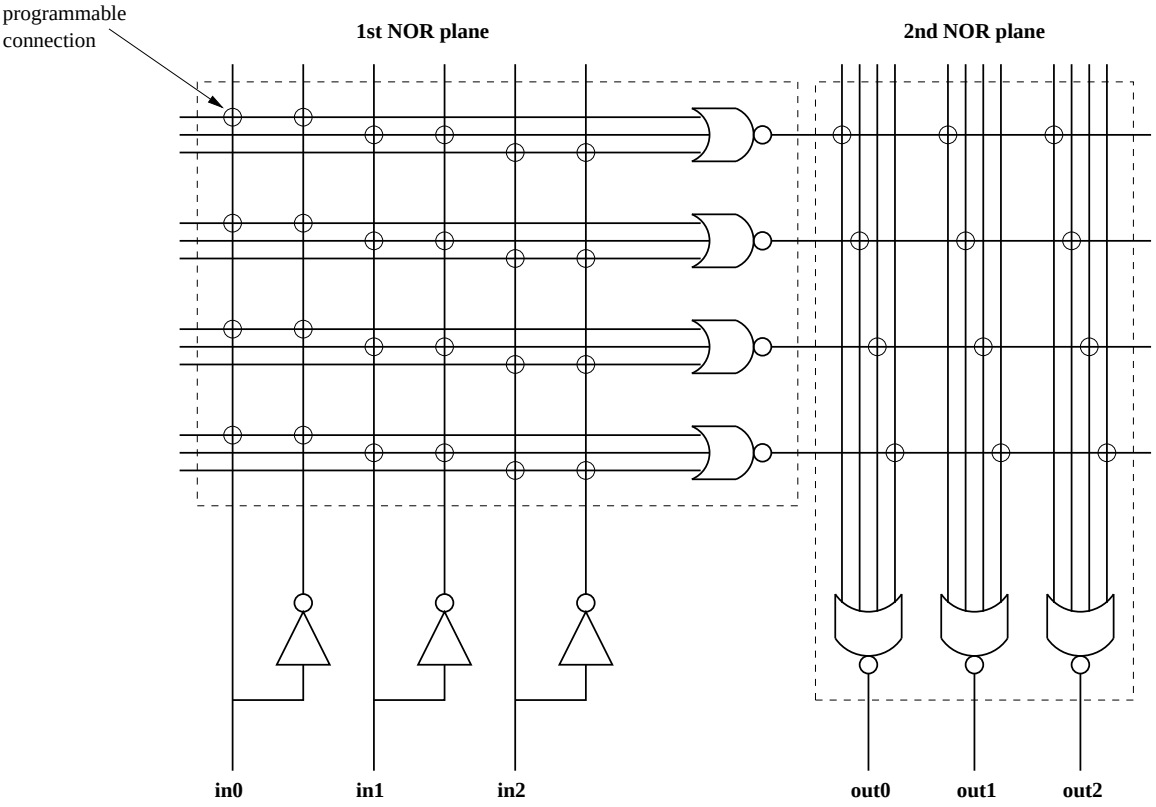

Figure 3.20: Example 3-input, 3-output NOR NOR PLA structure

the CAM performs a match operation and the SRAM uses its special logical wordline for a wired-NOR read.

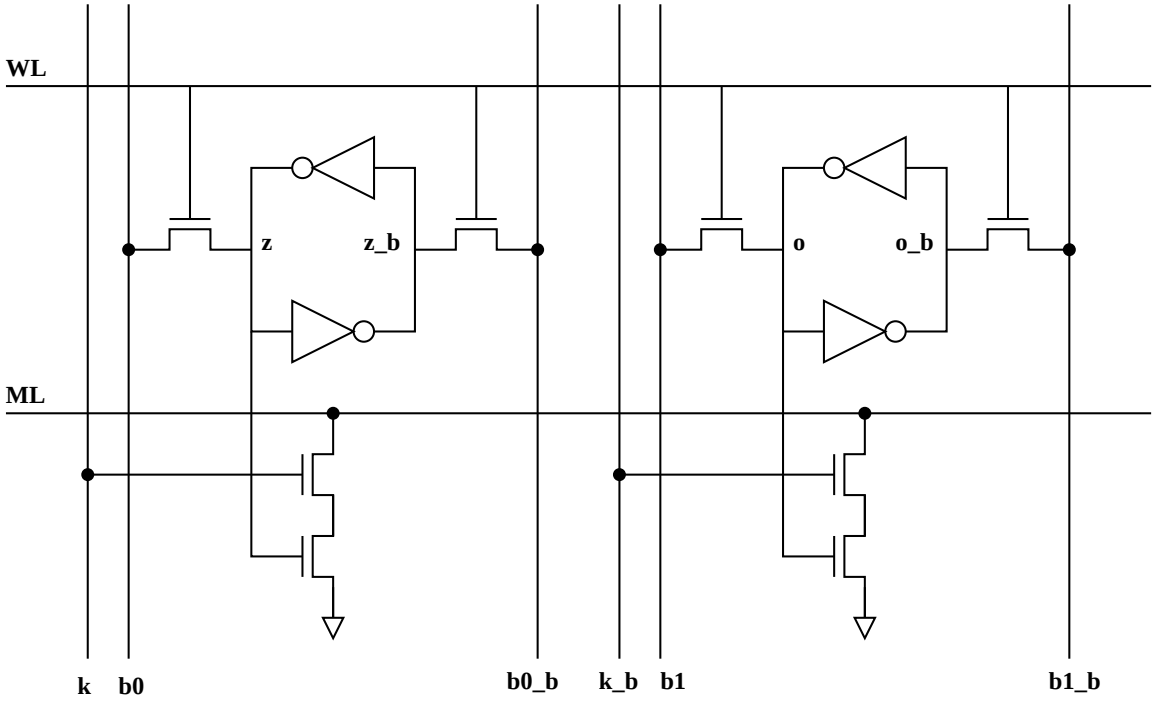

Figure 3.21: Ternary CAM trit cell

The NOR-style ternary CAM uses self-resetting matchlines, and the activation of the second NOR plane is timed using a replica matchline. Each CAM trit cell consists of two 6T storage cells and two NMOS pull-down stacks (Figure 3.21). The CAM's relatively small size allows for full-swing matchlines. Each row of trit cells forms a NOR gate with the matchline as the output. The ternary CAM cells pull down the matchline on a mismatch, and after a delay, self-reset circuits restore any mismatched matchlines to Vdd. The self-resetting matchline scheme will save power over a global matchline precharge scheme provided the configurations have relatively low matchline activity factors. Table 3.6 shows the correspondence between the stored values and logical meaning.

The activation of the second NOR plane is based on a replica matchline. One of the challenges in a NOR-style CAM design is to know when the matchlines have settled. In the rows that match, the matchlines remain high, but in the rows that mismatch, the matchlines

Table 3.6: Ternary CAM stored value meaning

z	o	Value	Action
0	0	*	Match on all
0	1	1	Match on 1
1	0	0	Match on 0
1	1	Null	Mismatch on all

are pulled low by some number of CAM cells. The falling delay of the matchline varies depending on the number of cells in the row that mismatch. By using a replica matchline that only mismatches on one cell, we replicate the worst-case matchline pull-down delay. Because both the actual matchlines and replica matchlines are full-swing signals, there is no signal amplification needed between the actual and replica matchlines as in SRAM replica bitlines [35].

The replica matchline generates a positive pulse on the *rml_b* line that mimics the delay of the worst-case matchline going low on a mismatch. The interface gate, a static CMOS NAND followed by an inverter, ANDs together the matchlines and *rml_b* to generate the logical word line signals *LWLs* which go to the SRAM array. Figure 3.22 shows a timing diagram of the ternary CAM and interface gate operation.

To ensure that the *LWLs* of all mismatched words do not erroneously glitch high, the matchline low time must be long enough to fully encompass the *rml_b* positive pulse. We designed the replica matchline to be 1 FO4 delay slower than the worst-case real matchline delay to ensure that rising edge of the *rml_b* positive pulse occurs after the falling edge of even the slowest falling matchline. We also must ensure that the matchlines do not reset too early, thus causing a glitch on the back-edge of *rml_b*. In the worst-case, a row could mismatch on all cells and thus the matchline would be pulled low as quickly as possible starting the matchline self-reset chain as early as possible. The matchline self-reset delay must be long enough to ensure that the matchline low time will still fully encompass the *rml_b* pulse under these conditions.

On the matchlines, we want both a long pulsewidth (*i.e.* long low time) and fast cycle time. If we used a standard self-reset circuit (Figures 3.23 and 3.24), in order to get a long

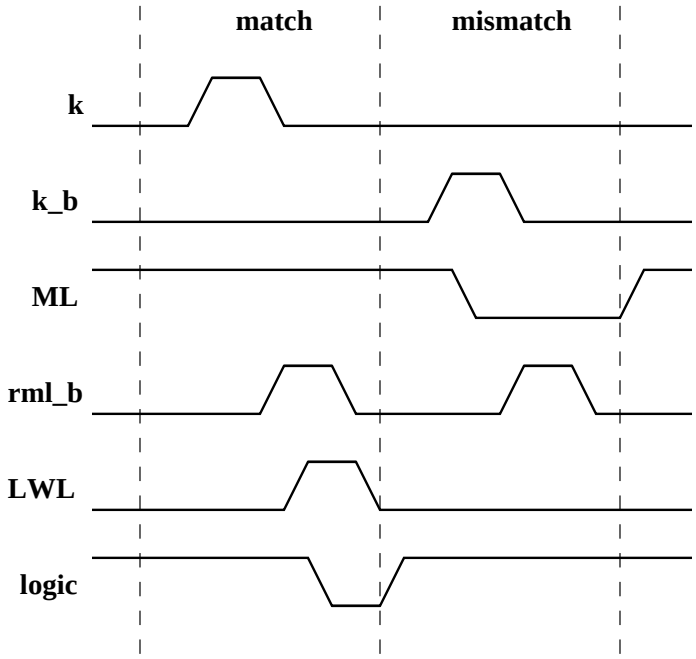

Figure 3.22: PLA ternary CAM timing diagram

pulsewidth, we must unnecessarily extend the cycle time to avoid a drive fight. After the matchline has been reset, we must traverse the delay chain a second time before *M1* is turned off and we can pull-down the matchline again. If we do not wait, then there will be a drive fight between *M1* and the CAM cell(s) pulling the matchline low. The second traversal of the delay chain uses the opposite transition from the first traversal. For a long pulsewidth and short cycle time, we need to skew the delay chain for a slow assert edge, to get a long pulsewidth, and a fast de-assert edge, to get a fast cycle time. One way to do this is to skew the P/N ratios of the inverter in the delay chain. However, this causes extremely slow slew rates on the slow transition which burn short-circuit current and are susceptible to noise.

Instead, we replace the second-to-last inverter with a NOR gate (Figures 3.25 and 3.26). The delay chain behaves as normal during the assert transition. However, on the second traversal, once the matchline reaches the logic threshold of the NOR gate, the NOR gate fires, skipping over the first two inverters. This shortens the delay chain de-assert time to two gate delays: one NOR gate and one inverter. We leave in the NOR gate and final

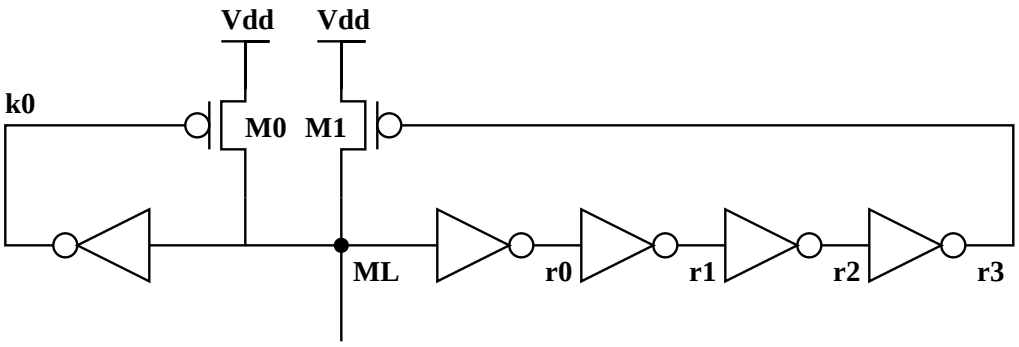

Figure 3.23: Normal self-reset circuit

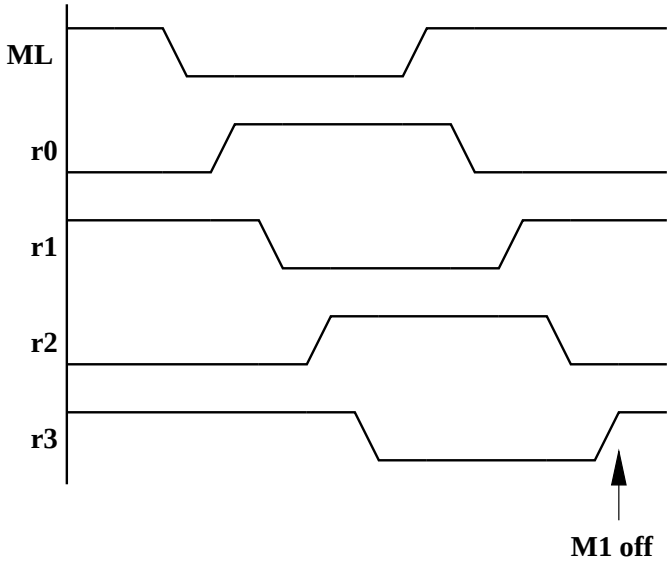

Figure 3.24: Normal self-reset circuit timing diagram

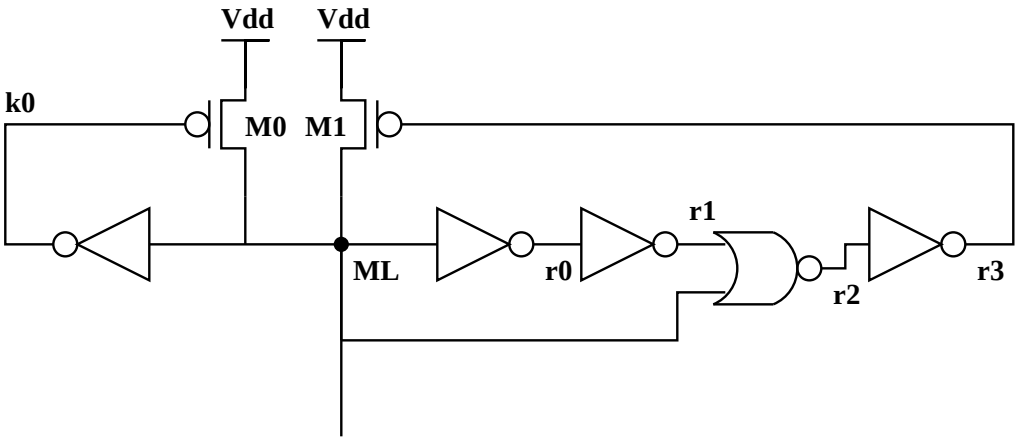

Figure 3.25: Fast reset off self-reset circuit

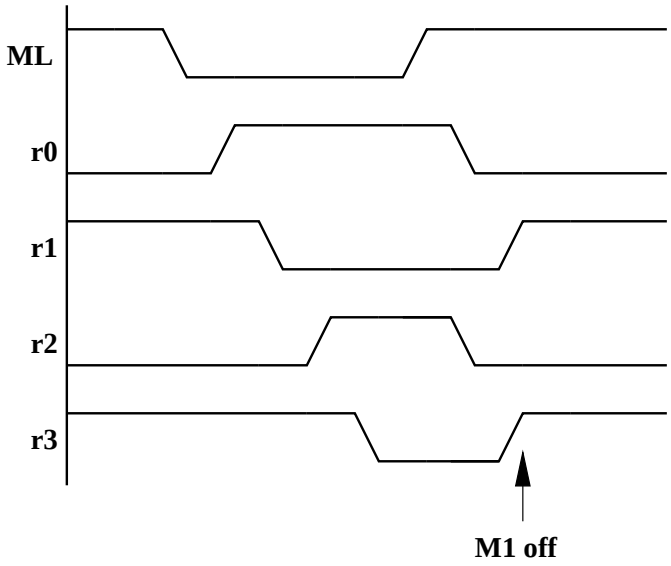

Figure 3.26: Fast reset off self-reset circuit timing diagram

inverter to allow the matchline time to reset all the way to Vdd from the NOR gate logic threshold. There is an additional keeper device (*M0*) on each matchline to ensure that the matchline reaches Vdd. By using this technique we can have a long pulsewidth, fast cycling matchline, while maintaining fast edges on all transitions of the delay chain.

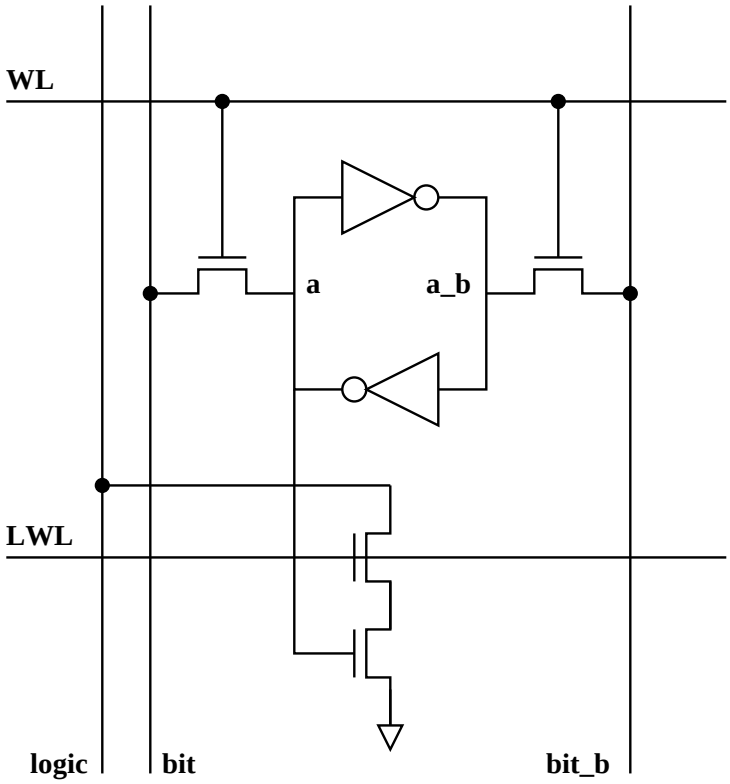

Figure 3.27: PLA SRAM cell

The *LWLs* generated from the matchlines and *rml_b* activate the second NOR plane implemented as an SRAM with a special logic output line. Each bitcell of the SRAM array (Figure 3.27) has a special logic line (*logic*) that is shared among all cells in a column. Within a column, all cells that store a one form a wired-NOR of the logical word lines (*LWLs*), with the *logic* node as the output. *Logic* is a full swing signal which is precharged early in the cycle, during the CAM evaluation. The small number of *logic* lines and their light loading made the precharge overhead relatively small and a reasonable trade-off against the increased area of self-reset delay chains.

3.6.2 Peripheral Logic

In addition to the meta-data and associated support logic, we add three peripheral logic blocks to a basic SRAM array: the pointer logic in the address path for pointer operations, the comparator in the datapath for comparisons, and a write buffer in both the address and data paths for conditional writes on external conditions. With the aggressive cycle and access time targets for the mat, the implementation of even very logically simple structures such as these require advanced circuit techniques.

Pointer Logic

The pointer logic consists of two small SRAMs and an adder/subtracter (Figure 3.28). A dual-ported SRAM stores the pointer address values, and a single-ported SRAM stores the strides, one stride per pointer. The "address" input to these memories is the pointer number specified by the operation.

On a pointer operation, the pointer SRAM reads out the address corresponding to the named pointer and sends it to the main decoder. The stride SRAM simultaneously reads out of the associated stride. The adder/subtracter calculates updated pointer address value from the pointer address and stride. We then optionally write the updated pointer address back into accessed pointer stored in the pointer address SRAM.

We dual-port the pointer address SRAM to allow simultaneous pointer address read and write-back of an updated pointer value. The SRAM properly handles the write-through case when the same pointer is simultaneously updated and read by outputting the updated pointer address value to the read. This allows us to do back-to-back pointer operations using the same pointer number. Because the strides do not need to be updated, the stride SRAM is only single-ported. The pointer address values and strides are read- and write-able via the configuration read and write instructions.

For a simple FIFO, we only need to store two pointers, the head and the tail, with each of their strides set to 1. However, by storing more than two pointers per mat, we can enable using a single mat as multiple FIFOs. For example, the testchip implementation stores four pointers per mat, which allows us to hold two FIFOs in a single mat. We could use pointers *P0* and *P1* for FIFO 0 and *P2* and *P3* for FIFO 1. By using a stride of 2 and offsetting the

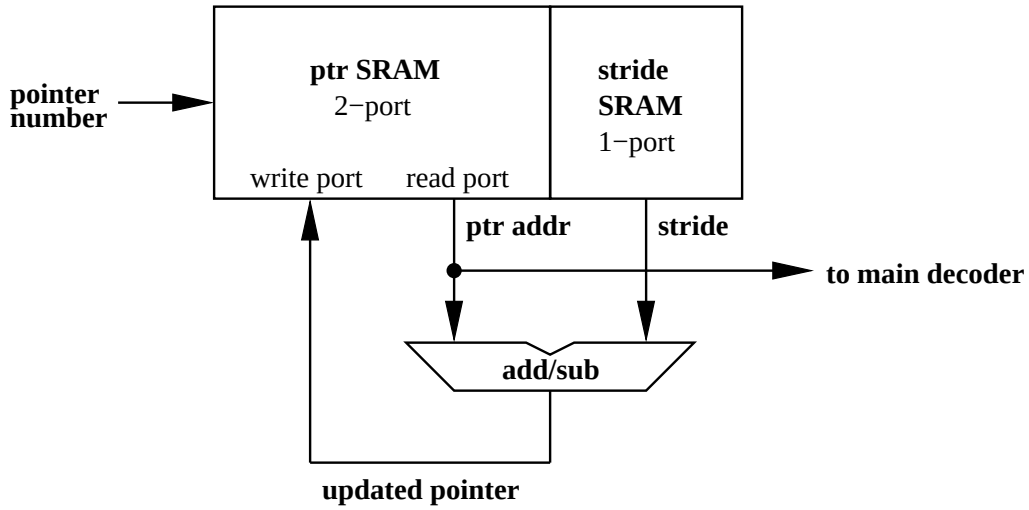

Figure 3.28: Pointer logic block diagram

starting head/tail values by one, we can interleave FIFO 0 and FIFO 1 in a single mat, with FIFO 0 occupying all even addresses, and FIFO 1 occupying all odd addresses. Figure 3.29 illustrates this configuration.

We can also enable FIFOs larger than one mat by storing a pointer address value that is longer than necessary to address the words of one mat. All requests to the FIFO are multicast to all mats in the FIFO, and the mats range check the upper bits of the pointer to determine which mat the request should access. All mats keep their pointer logic in lock-step, so even if the mat is not accessed, the pointer logic is updated. The number of extra bits in the pointer addresses determines the maximum FIFO size. For example, a pointer that has two extra bits allows for a maximum FIFO size of four mats. Chapter 4 discusses how the interconnect structure can multicast to all the mats in the FIFO. Figure 3.30 shows a single FIFO that spans four mats.

Maskable Comparator

The maskable comparator implementation is relatively simple, differing only slightly from regular comparators, on which there is a large body of work [101][102][103][104]. Our comparator differs in that it can mask out select fields of the data, treating them as don't

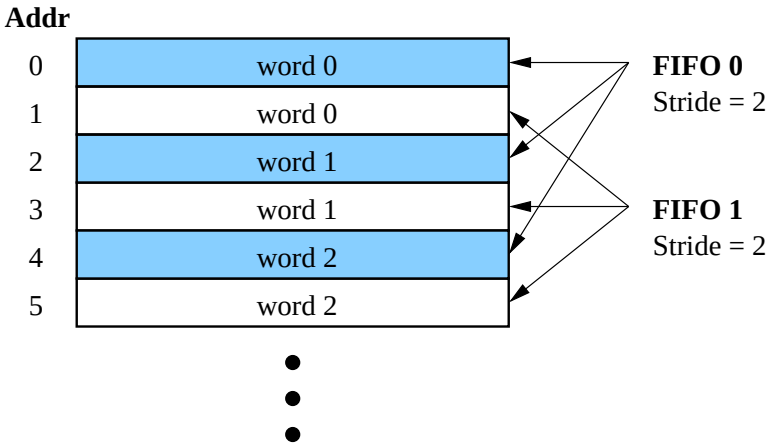

Figure 3.29: Storing two FIFOs in one mat

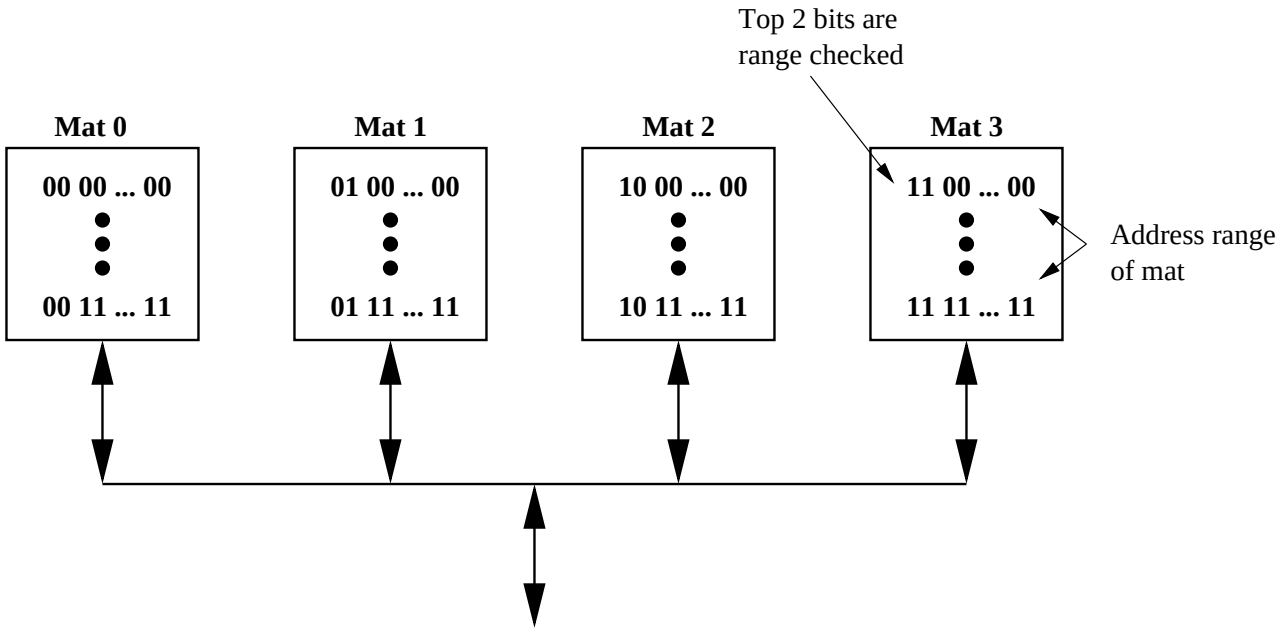

Figure 3.30: A FIFO spanning four mats

cares. We can mask out any combination of the meta-data bits and the main data as a chunk. The comparator (Figure 3.31) uses a single mask bit *mask[0]* for all of the main data, but individually masks each bit of meta-data . We implement the final wide fan-in AND gate as a tree of smaller fan-in gates using the principles of logical effort [32].

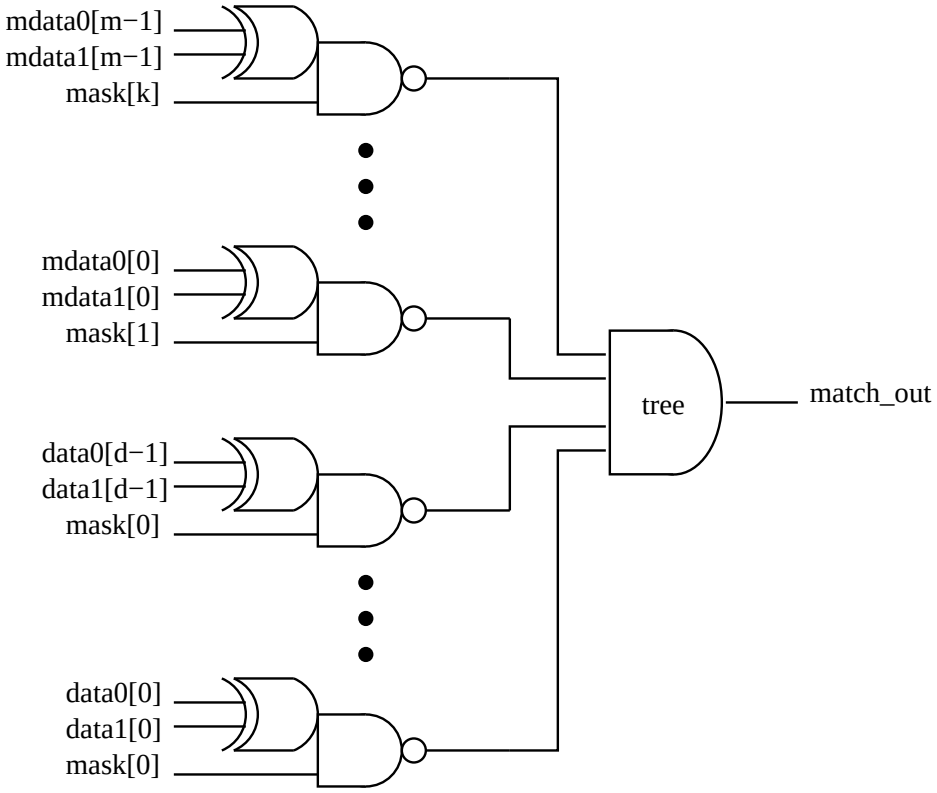

Figure 3.31: Maskable comparator gate-level diagram

The comparator is implemented as a tree of dynamic gates. We could embedded the initial comparison XOR gate in the bitline column muxes, but this introduces more complexity in the column mux circuits. Depending on the implementation, this could also introduce another series device in the bitline path.

Write Buffer

To enable writes contingent on an external condition, we add a write buffer in both the address and data paths. On a conditional write, the write buffer stores the incoming write

address, data, and opcode. When the external condition evaluates, the write buffer invalidates the entry if the condition fails, or allows the entry to write into the main data array in a pipelined fashion if the condition is met. This occurs in the same way that some cache designs pipeline writes to allow single cycle cache write hits [105] as described below.

In a traditional cache implementation, a cache write operation must serialize the tag check and the write into the data array. Because the write into the data array is an unrecoverable change of state, we cannot execute it until we know that the access is a cache hit. The serialization of the tag check and data write increases the latency of a cache write significantly.[3]

A well-known technique to hide the data write dependence on the tag check is to pipeline the cache write into two logical stages: tag check, data write. Because not every access is a write, the operations are not pipelined on a per cycle basis, but rather on a per write basis. This technique requires a small write buffer in the data array to store the incoming write data until its tag check has completed. The number of storage locations in the write buffer depends on the mat implementation, specifically on the latency of the tag check. The write buffer must be searchable to ensure that any reads or compares to the stored write address that occur before the data can be written into the main array behave properly.

3.7 Summary

In this chapter, we explored the motivation for a reconfigurable memory system and our proposed memory architecture. This chapter also examined the design of the reconfigurable memory mat that forms the core of the memory system. By adding the meta-data bits and a few peripheral logic blocks to a simple RAM array, we can create a flexible memory block that can emulate a portion of a cache, a FIFO, or scratchpad memory. We have proposed an implementation of that architecture using full-custom circuit design techniques for high performance and efficiency. The next chapter will detail the remaining portions of the

[3] For highly set-associative caches, designers take advantage of this dependence to save power. They intentionally serialized the tag check with the data access (even for reads) to reduce the number of data arrays that are activated from *n* (number of ways) to just 1 (hit) or 0 (miss in all ways).

reconfigurable memory system: the inter-mat control network, the memory to processor interconnect, and the processor interface logic.

Chapter 4

Interconnect Networks

As seen in the previous chapter, because memory structures use similar building blocks, we can use a generalized memory mat to form the core of our reconfigurable memory system. However, memory systems do differ significantly in the way that they connect the memory blocks to the computation and to each other. For example, in a cache, each memory request goes to both the tag and data array, and the tag array communicates hit/miss information to the data array. However, in a simple scratchpad memory, each memory request only goes to one memory block, and the memory blocks that make up the scratchpad never communicate with each other. So, for our configurable memory system that can emulate multiple types of memories, we need a configurable interconnect network between the mats and the processor and between the mats themselves.

The interconnect network must handle two types of communication, mat-to-mat and mat-to-processor, that have quite different requirements and characteristics. Mat-to-mat communication is primarily one-to-many (*e.g.* a tag mat sending its hit/miss information to multiple mats in the cache data array), and the width of this data is typically very narrow, consisting of only one or two bits. Additionally, the mat-to-mat communication is unidirectional, requiring only outgoing data pushes without any replies from the recipients. The latency, however, must be extremely low, because the communication can be in the critical path of the memory structure.

On the other hand, mat-to-processor communication is bi-directional and many-to-many. There can be multiple concurrent outgoing requests from the processor, all to different mats, as well as multiple returning replies from the mats. The requests and replies are wide, composed of data, meta-data, and possibly opcode and address fields. While the latency of the mat-to-processor communication must be low, because it can be in a critical loop that determines the minimum processor cycle time, it still can be on the order of a cycle.

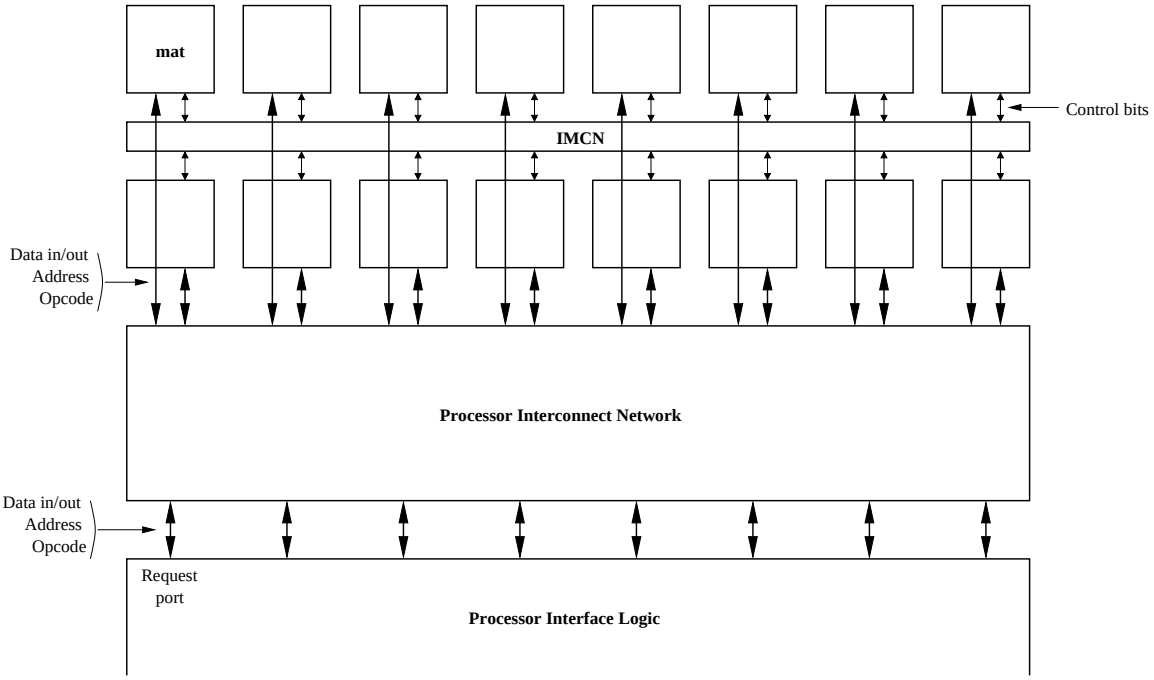

Figure 4.1: Interconnect overview

Because the mat-to-mat communication and mat-to-processor communication differ significantly in their characteristics and requirements, we separate the interconnect into two networks: the mat-to-mat *inter-mat control network* and the mat-to-processor *processor interconnect network*. Figure 4.1 shows an overview of the interconnection networks. We implement the inter-mat control network as multiple segmented buses and the processor interconnect network as a pair of crossbars. However, before delving into the specific architecture and implementation of the networks, we first review some relevant issues regarding

interconnection network design.

4.1 Interconnection Network Design

For many years, interconnection networks have been used to connect computers and discrete processors [106][107]. Today's die capacities, however, allow multiple processor cores and large module blocks such as memories to fit on a single die. Thus, researchers have proposed on-die interconnection networks to connect processors and large modules together rather than hardwired global routing [108][109][110][111]. These networks offer a more structured communication environment than custom global routing, which allows for more design re-use and easier application of custom circuit techniques for lower power and higher performance.

While the concept of a "network on a chip" to connect large modules together has gained popularity, the use of interconnection networks between processor functional blocks has been quite limited [112]. Most functional blocks are still connected together with fixed routing or buses. While buses do offer a some communication flexibility, they do not perform well under communication patterns with multiple concurrent requests [113]. For increased request concurrency, some designs use split transaction buses where the requester releases the bus while waiting for the reply [105][85], and other transactions can use the bus for requests or replies in the mean time.

Another technique to improve bus bandwidth is to physically segment the bus and allow concurrent transactions on different segments [114][115][116]. This technique can achieve high concurrency for communication patterns where units communicate locally. Bus segmentation can also reduce the power dissipation of the bus by limiting the portion of the bus that is activated.

Buses are very area efficient, but as the number of concurrent transactions increases, they are no longer the optimal interconnection topology. A crossbar interconnect topology handles multiple concurrent transactions better than a bus-based system [113][117] because it is a fully non-blocking network. Any of the inputs can be connected to any of the outputs as long as both the input and output are free. A crossbar is a one-hop network (*i.e.* has no intermediate stages between the input and output) and thus can have low latency. As per

convention, we will denote a crossbar with *N* inputs and *M* outputs as an *N* x *M* crossbar, with each possible connection point known as a crosspoint. An *N* x *M* crossbar would have *N*M* crosspoints. Figure 4.2 shows an example 4 x 6 crossbar.

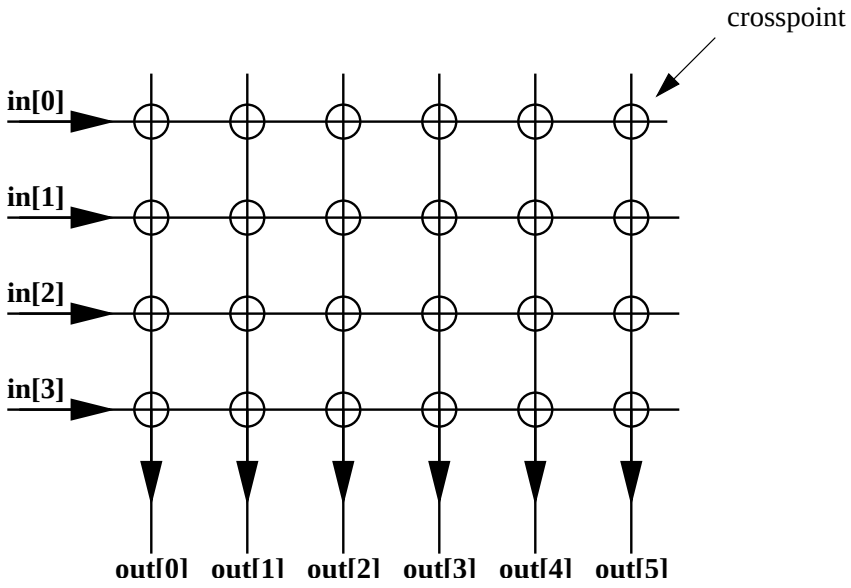

Figure 4.2: 4 x 6 crossbar

Crossbars topologies have been used in many networking switching architectures [107][106], and most previous crossbar implementations have either been stand-alone designs for networking switches [118][119][120] or on-chip interconnection networks connecting processors to main memory [121][122][123][124]. Crossbars that interconnect individual functional blocks are much rarer [112][125][126], although, one very common crossbar-like interconnection structure between functional units is a pipelined processor's functional unit bypass network [105][127][128].

While a crossbar does support multiple concurrent requests, conflicts can occur if more than one input requests the same output. Telecom crossbars avoid conflicts using complex input queues, arbiters, and scheduling logic [129][130][131][132]. The arbitration and scheduling add to the latency, area, power, and complexity of the design. The delay from input to output is now dependent on the request pattern and is non-deterministic. This is tolerable in networking switching applications, but for fixed pipeline, inter-functional unit

networks a deterministic latency that is as low as possible is desirable. An alternative to dynamic scheduling is to statically schedule the crossbar. This avoids the added latency and complexity of arbitration, while still allowing multiple concurrent requests [122].

Despite its larger area, a crossbar can be more energy efficient per transaction than a segmented bus architecture, due to the large amount of concurrency available [137]. One way to significantly decrease the energy consumption of a crossbar is to use low-swing interconnect techniques [56][55][133][134], because a crossbar has a large number of long, high-capacitance wires. To further reduce the energy dissipation, we can segment a crossbar to decrease the wire capacitance driven [135][124].

Besides crossbars, there are many other non-blocking network topologies and hybrid topologies that offer a different trade-offs between area efficiency, energy consumption, and performance [106][107]. For the inter-mat control network and the processor interconnect network, we chose topologies that seemed most suitable to the communication patterns on those networks. Next, we will examine an implementation for the inter-mat control network.

4.2 Inter-mat Control Network

The inter-mat control network allows the mats to pass a few bits of control information to each other. For example, in a cache configuration, the tag mat must pass the hit/miss information to the data mat, so that the data mat knows whether or not to abort its operation. On a hit, the data mat proceeds with its operation, but on a miss, it aborts. We implement the IMCN as multiple, 1-bit, segmented buses. Figure 4.3 shows an example implementation with four buses with four mats per segment. The IMCN is well suited to this implementation, because it is a network with uni-directional, one-to-many communication. The bus segments are set at configuration time via configuration registers.

We use full-swing wires in the IMCN, for a number of reasons, First, it is a relatively short-haul network, and the overhead latency of the low-swing driver and receiver are not tolerable. Second, because the latency is a fraction of a cycle, there is no convenient clock edge off of which to trigger the sense amplifiers. Finally, the IMCN does not contribute significantly to the overall power dissipation of the reconfigurable memory system, so the

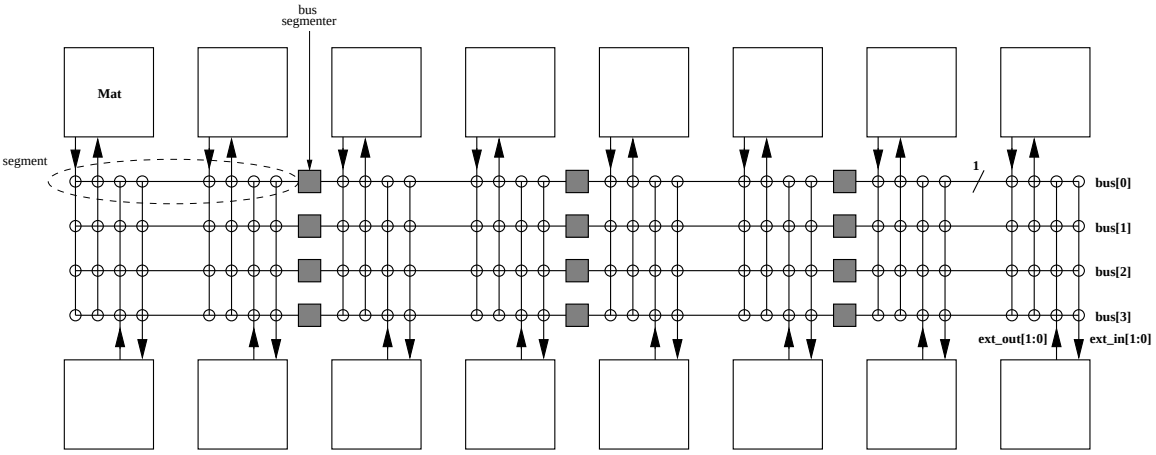

Figure 4.3: Inter-mat control network

overall savings from reducing the IMCN power would not be significant.

Each mat has a two-bit output to the IMCN, *ext_out*, and a two-bit input from the IMCN, *ext_in*. A number of signals are multiplexed to generate the two bits of *ext_out* as seen in Figure 4.4. Similarly *ext_in* is demultiplexed to a number of locations as shown in Figure 4.5 The inputs and outputs are statically configured via configuration registers.

The IMCN bus drivers can implement a wired-OR function on the bus. The driver is a pull-down only driver and thus can form a wide OR gate with any other active drivers on the bus segment. Figure 4.6 show the bus driver and pull-up circuit. The wired-OR functionality is useful for aggregating control information. For example in a multi-way cache configuration, by OR-ing the hit/miss signal from every way, we can generate the global hit/miss for the cache.

The pre-charged IMCN bus line is vulnerable to signal coupling, but the weak keeper helps mitigate coupling problems. If the IMCN lines are still deemed too vulnerable, shield wires could be routed next to each IMCN line to reduce coupling effects. Given the small number of IMCN lines, the additional area penalty of the shield lines is not too great.

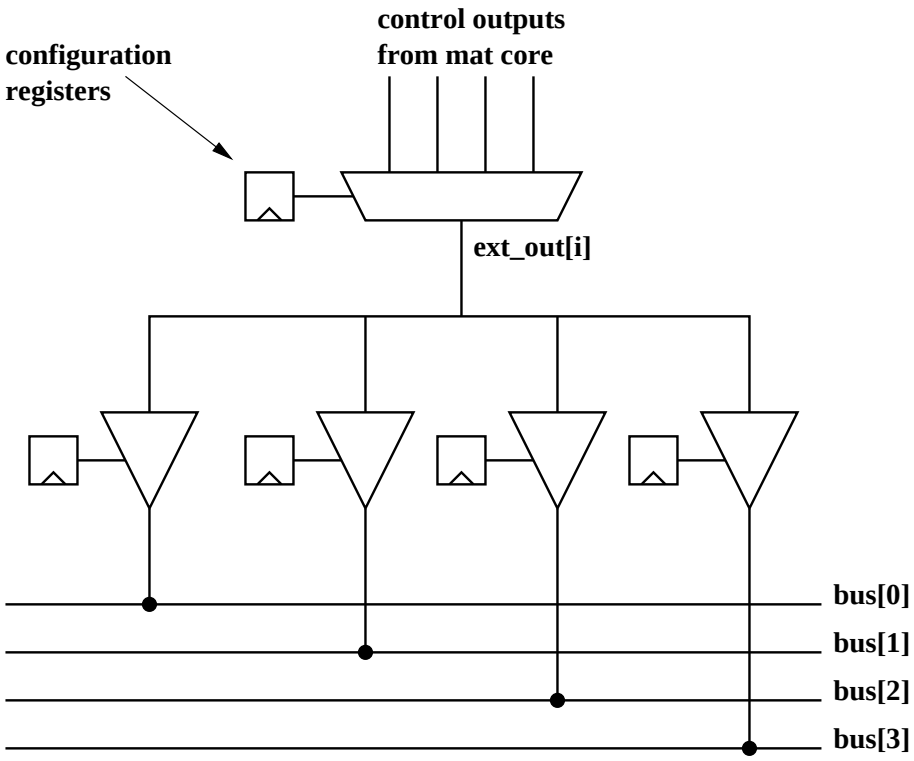

Figure 4.4: *Ext_out* logic

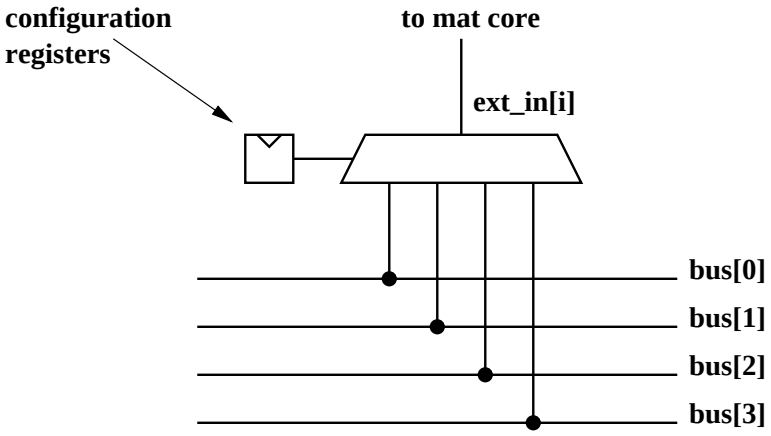

Figure 4.5: *Ext_in* logic

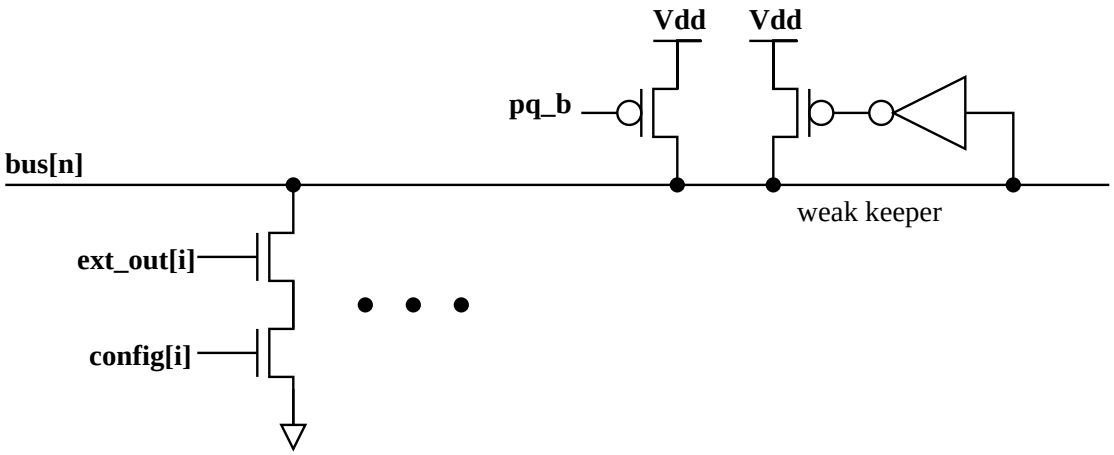

Figure 4.6: IMCN wired-OR driver and pull-up circuit

4.3 Processor Interconnect Network

In our reconfigurable memory system, the processor interconnect network takes the place of the fixed interconnection between the computation and the memory in hardwired systems. The processor issues memory requests from request ports in the interface logic. Any port can send a request to any memory mat. The mat always returns the reply to the port that issued the request. Section 4.3.4 describes the interface logic request ports more fully.

We implement the processor interconnect as a pair of uni-directional crossbars as shown in Figure 4.7. For the sake of clarity in the figure, the crossbars are shown as distinct, but in the implementation, they are interleaved into one block. The *request crossbar* routes the requests from the processor ports to the memory mats, and the *reply crossbar* routes the replies from the mats back to the requesting processor port.

We chose to use crossbars for the processor interconnect, because they support multiple concurrent transactions, have low, one-hop latency, and offer high flexibility in the interconnections. We made the design decision to use two uni-directional crossbars rather than a single bi-directional crossbar to simplify both the circuit design and system architecture at the cost of some area. The circuit design for a uni-directional crossbar is simpler, and we avoid the additional delay and energy from driving the parasitic capacitance of the drivers and receiver of the unused signaling direction. At the system level, a bi-directional crossbar would require more careful memory access scheduling, especially for requests that both push and pull data, such as a compare. This could require arbitration for crossbar and cause the memory access latency to be non-deterministic, further complicating the processor interaction with the memory system.

4.3.1 Request Crossbar

The request crossbar is a dynamically-routed, multicast-capable crossbar responsible for routing requests from the processor ports to the memory mats. Each request port can send a request to any mat or group of mats. Every request has a *mat ID* field that determines the destination of the request. This routing based on the *mat ID* can be used as the high-level decode for multi-mat memory structures, similar to the block selection mechanism of large SRAMs [53][25][31]. Figure 4.8 shows a block diagram of the request crossbar with each

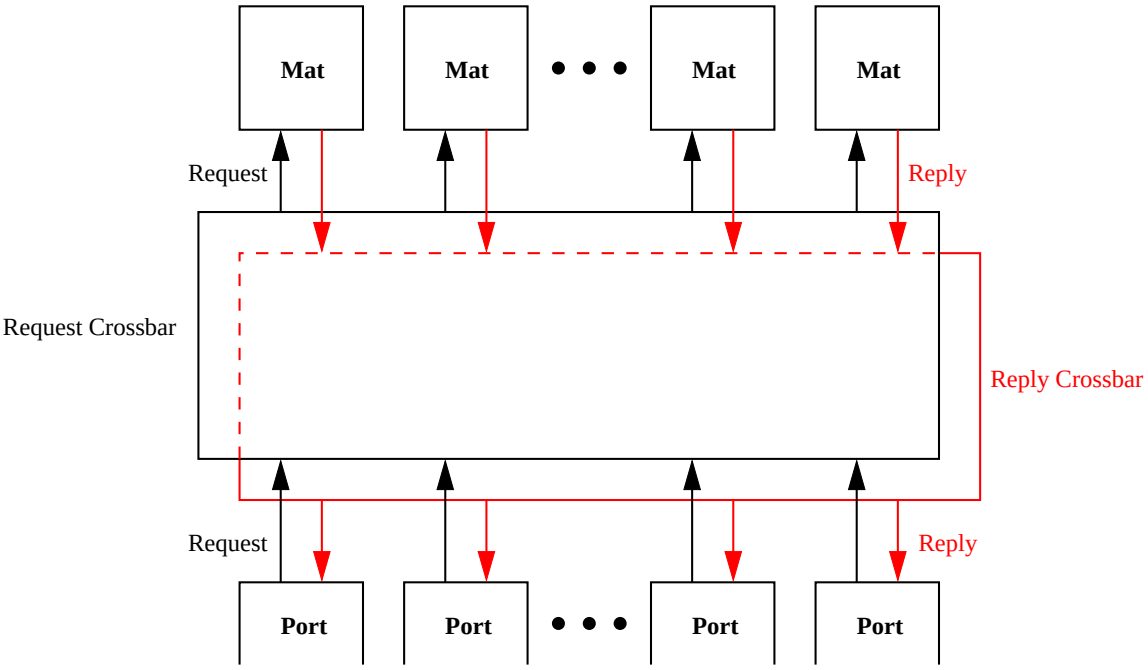

Figure 4.7: Processor interconnect overview

crosspoint detailed in Figure 4.9. Table 4.1 lists the request crossbar I/O signals.

The *mat ID* and *mat ID mask* determine the request destination mat(s). The *payload* is the request itself that goes to the mat. It contains all the mat input fields from the processor: *opcode*, *address*, *mask*, *mdata_in*, and *data_in*. The *valid* bit indicates whether or not the outgoing request is valid. The *reply* bit alerts the reply crossbar scheduler that the request has reply data. The crosspoint will then schedule a reply data packet transfer on the reply crossbar.

To avoid request conflicts, we statically schedule which ports have access to which mats in the processor interface logic. The request crossbar still dynamically routes the requests based on the *mat ID* field, so the ports have some freedom in which mats to send request to and can use the crossbar for high-level decoding in multi-mat memory structures. However, there is no need for queuing, arbitration, or scheduling logic, and the delays through the crossbar are fixed and deterministic.

An example of static mat scheduling would be splitting the access to a multi-processor

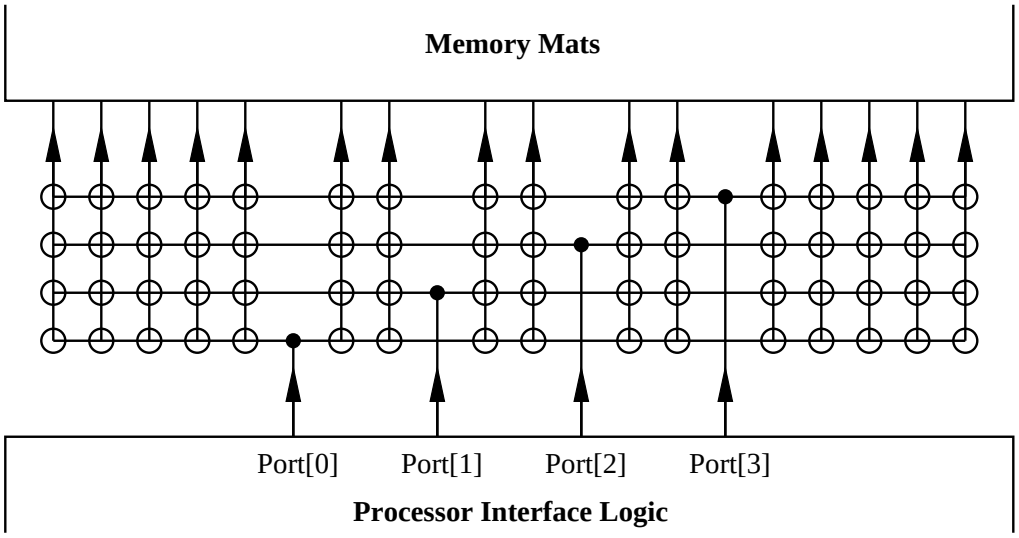

Figure 4.8: Request crossbar

Table 4.1: Request crossbar I/O signals

Name	Width	Description
mat ID	i	Which mat to send the request to?
mat ID mask	i	Allows for wildcarding in the mat ID
payload	p	Request to the memory mat
valid	1	Is this a valid request
reply	1	Does this request have return data?

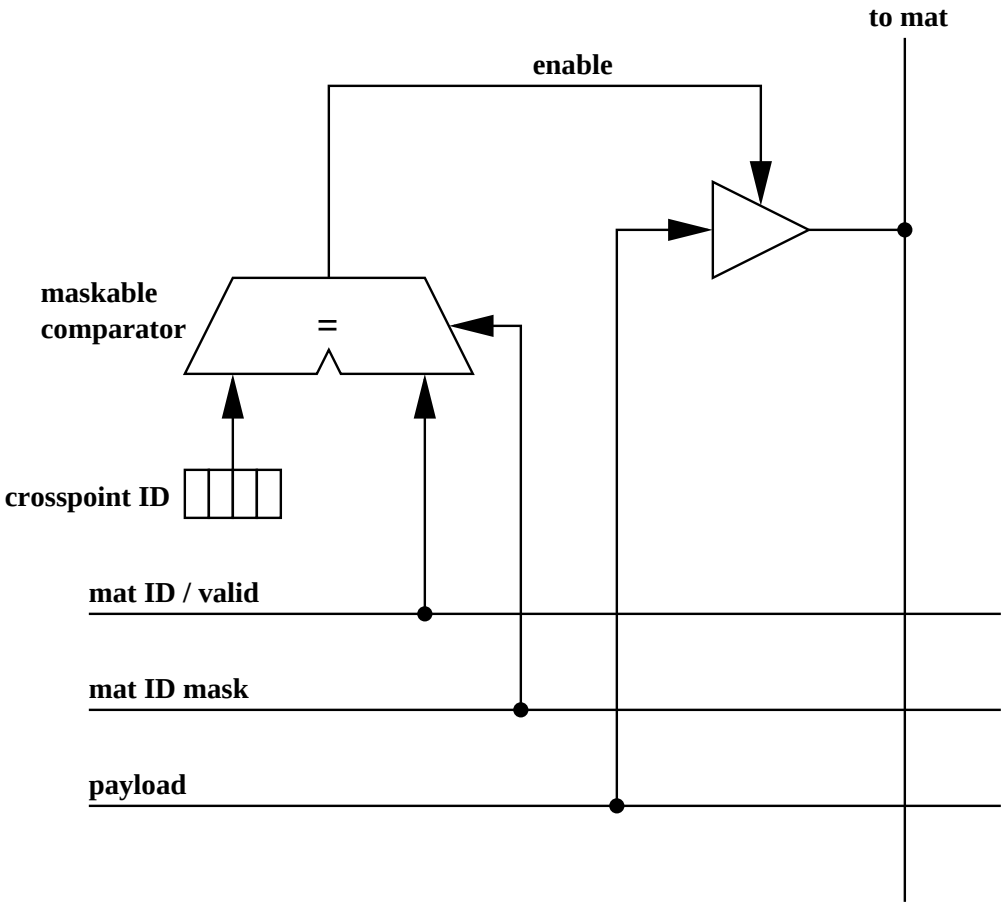

Figure 4.9: Request crossbar crosspoint

cache tag between the normal access and the snoop access. Every even cycle would be dedicated to the local processor cache access and every odd cycle to the snoop access. If we ran the memory and crossbar at twice the rate of the processor core, the processor would still get one cache access per processor cycle. This is an application of virtual multi-porting described in Chapter 3.

The downside is that static mat scheduling potentially wastes bandwidth on low usage requesters, and we need to ensure that there is enough bandwidth to the memories to satisfy all requesters under the static allocation policy. If the memory mats and crossbar have more bandwidth available than most applications need, statically allocating the bandwidth, although wasteful in some cases, may not adversely affect the application performance. Also, because the processor interconnection network emulates the static interconnect found in hardwired memory systems, we do not expect configurations to require dynamic scheduling. However, using static mat allocation is not fundamental to the design, and the reconfigurable memory system could be implemented using dynamic scheduling at the cost of increased design complexity.

4.3.2 Reply Crossbar

The reply crossbar routes memory replies from the mats back to the processor ports. The crossbar always routes the reply back to the requesting port. Thus, we can schedule the reply crossbar based on the request routing and which requests will have replies as indicated by the *reply* bit in the request. Because the mat latency is fixed, the scheduling of the reply crossbar is simply a time delayed version of the request schedule. Figure 4.10 shows an overview of the reply crossbar, and Figure 4.11 details a reply crossbar crosspoint. Table 4.2 lists the I/O signals for the reply crossbar.

The contents of the reply packet depend on the request operation. If the request is any form of a read (*e.g.* read, compare, read-modify-write), then the reply contains data and meta-data. Writes and gang operations do not return data or meta-data. Every request operation returns three control bits, *valid*, *match*, and *complete*, in the reply packet.

The request crossbar can multicast requests to multiple mats, but there can be only one valid reply to a multicast request. The reply crossbar multiplexes the outputs of the accessed

Table 4.2: Reply crossbar I/O signals

Name	Width	Description
data	d	Main data
mdata	m	Meta data
valid	1	Is this data valid?
match	1	Match output result
complete	1	General completion bit

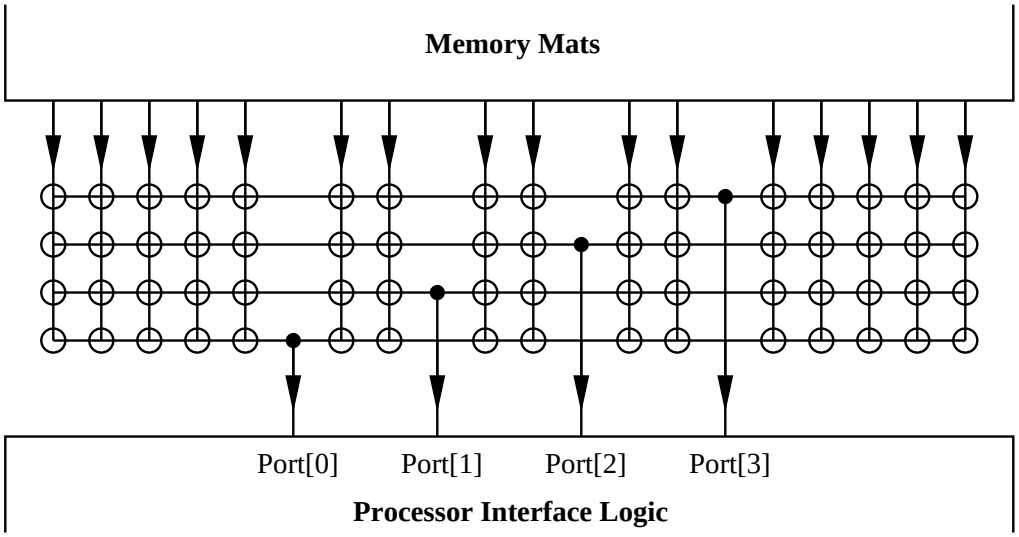

Figure 4.10: Reply crossbar

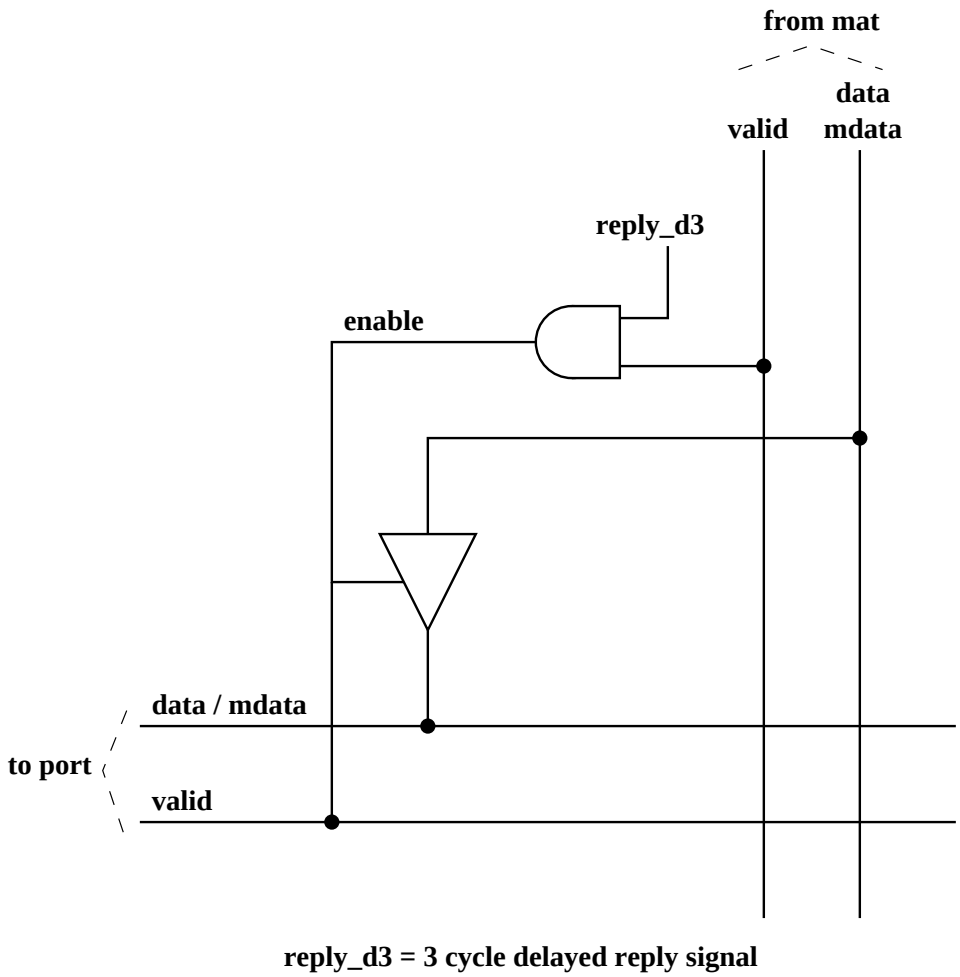

Figure 4.11: Reply crossbar crosspoint

mats and only send a single reply back to the processor port. The *valid* bit from the mats indicates which mat has the valid reply packet. By design there can only be one valid reply to a multicast request, and this must be guaranteed by the configuration of the mats. This type of multicast-request/multiplexed-reply combination is useful for configurations such as multi-way set associative caches. In such configurations, we multicast the memory request to all ways of the cache, but only want the reply for way that hits. If no way hits, no reply is received.

4.3.3 Implementation

For both the request and reply crossbar, we employ low-swing, differential signaling using NMOS only drivers and clocked sense amplifiers [54]. The NMOS driver circuit, shown in Figure 4.12, pulls one of the differential output lines to *Vdd_low* and the other line to Gnd based on the data input. *Vdd_low* is a special, low-voltage supply that can either be generated locally via DC-DC conversion [136] or taken in from the outside world. By using a lowered supply, rather than just limiting the swing, we can achieve a quadratic energy savings in swinging the wire [56]. We do then, however, need to route, decouple, and potentially generate the lowered supply voltage. The receiver sense amplifier is a modified StrongARM latch (Figure 4.13) [57]. Because we use a low-swing, differential signaling scheme, we require two wires to transmit each bit. We twist the differential lines and interleave them with power supply lines to reduce the differential crosstalk noise [56][39][40]

There are additional techniques for reducing the power of the crossbar such as segmentation [135], but we chose not to employ them in our design to keep the design as simple as possible and avoid any excess increase in the latency. There have been a number of works that explore the energy consumption of crossbars in much more detail [137][135][138].

Conceptually a crossbar has all inputs arriving from the sides and all outputs exiting on the top and/or bottom as in Figure 4.2. For our design, both the memory mats are above the crossbar and the processor ports are below the crossbar to achieve a rectangular shape for the entire reconfigurable memory. Figures 4.8 and 4.10 show the wiring arrangement needed to support this configuration.

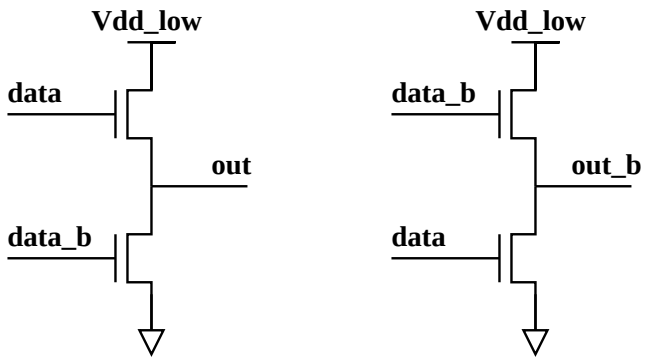

Figure 4.12: Low swing driver

We allocate an entire clock cycle for signals to traverse the crossbar. The drivers transmit the signals on the rising edge of the clock. The receivers flop them on the next rising edge. From circuit-level models of the crossbar, 10 FO4, 1ns in our target 0.18μm technology, is enough time to traverse the crossbar in either direction. A study of inter-functional unit crossbars shows that a similar 16-port, 32-bit crossbar design can maintain sub-1ns latency in a 0.25μm technology [112]. Chapter 5 provides more details on the testchip implementation of the crossbars.

4.3.4 Processor Interface

While the memory mats connect directly to the processor interconnect network, the processor launches requests into the processor interconnect network via the *processor interface logic*. The main function of the interface logic is to translate the memory address issued by the processor to the address space used by the interconnect network.

Hardware Address Space

In modern computing systems, there are often two sets of address spaces, virtual and physical [105]. The applications operate in individual virtual address spaces. Virtualizing the memory removes the burden of memory management from the application programmer, and allows multiple applications to more easily share the limited physical memory. The physical address is used to address the physical main memory.

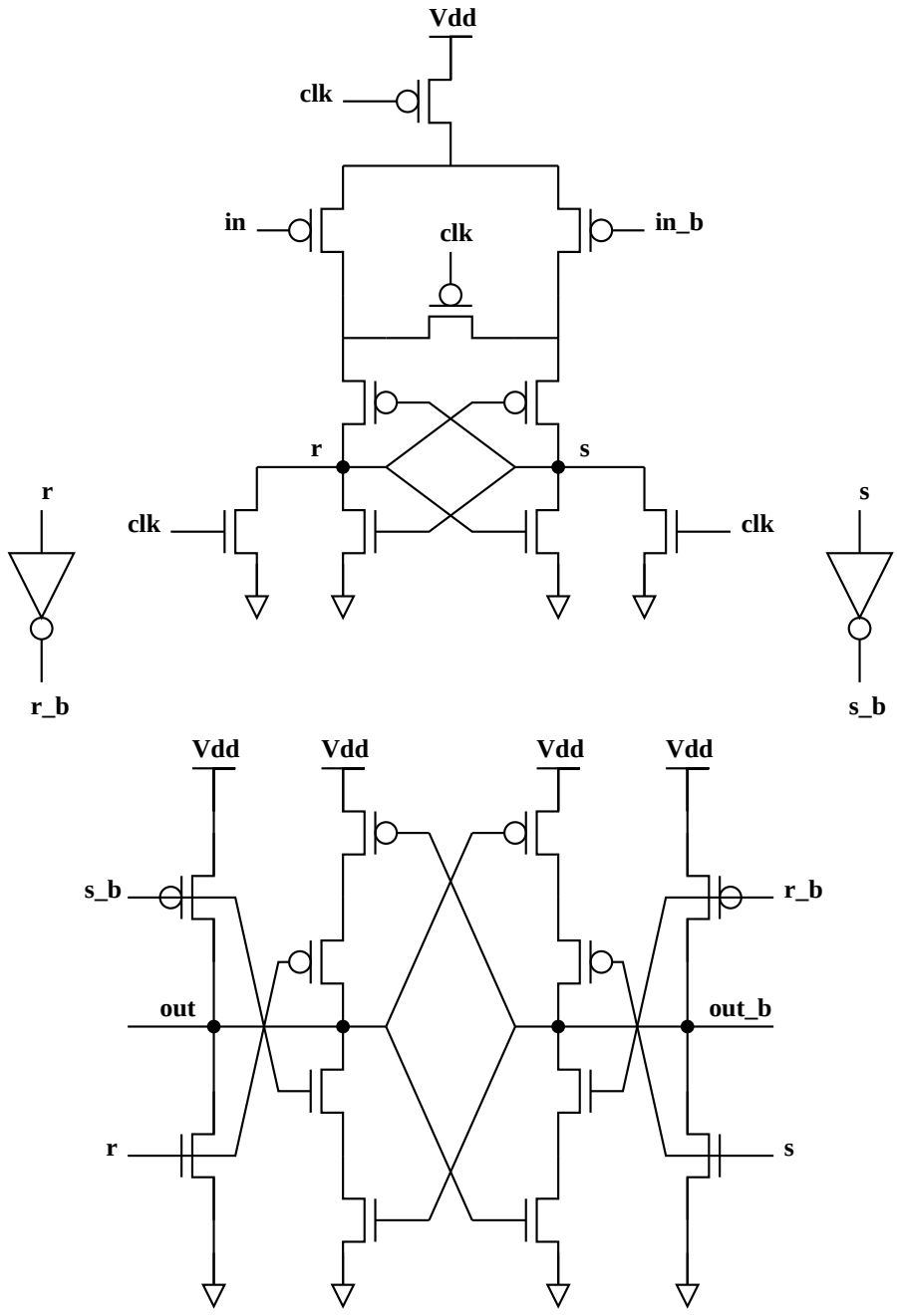

Figure 4.13: Low swing receiver - modified StrongARM latch

When the processor issues a memory request, the request names the desired virtual address location. This virtual address undergoes address translation to a physical address via a translation table which maps the virtual address space to physical address space. Because the translation table itself can be quite large, often used translations are cached near the processor in a look-up structure called the *translation look-aside buffer* (TLB). In a hierarchical memory system, the virtual to physical translation can take place anywhere in the hierarchy before the main memory.

Memory is broken up into fixed size chunks called *pages* or variable sized chunks called *segments*. Because the total amount virtual memory is larger than the physical memory, data in the virtual address space is stored on disk and paged into main memory by the OS when needed. The main memory can be thought of as a cache for the disk system, similar to how the on-die processor cache is a cache for main memory. While virtual memory is a useful for systems that run multiple concurrent applications, some simpler systems, such as some DSPs and embedded systems, only use a single address space. Their applications are responsible for memory management, but the overall system is less complex.

For our reconfigurable memory system, we introduce a third address space, the *hardware address space*, which is conceptually below the physical address space. Just as the physical address designates locations in the main memory, the hardware address names memory locations in the on-die reconfigurable memory system. The hardware address consists of two fields, the *mat ID* and *mat address*.[1] The processor interconnect network uses the mat ID to route requests to the proper mat. The mat decoder uses the mat address to select the desired word in the mat. In a hardwired memory system, the hardware address space is not necessary, because the local memory on the processor die is "addressed" implicitly via the hardwired interconnections between the processor and memory.

The basic issue is that the processor generates virtual addresses, but in order to access the correct local memory, the virtual address must be translated into a hardware address. Usually since the hardware is fixed, the mapping between the virtual address and the hardware address is hardwired into the hardware structures. However, for a reconfigurable system, the memory configuration is not fixed, and as a consequence the translation between

[1] In a tiled system with multiple mat arrays, there can also be a *tile ID field* that denotes which tile the memory location resides in.

the virtual and hardware address spaces is not fixed. Thus, we need a block that performs the virtual to hardware address translation. This address translation is fairly simple when compared to the virtual to physical address translation.

Address Splitter

From the virtual address issued by the processor, we need to generate two fields to properly access the reconfigurable memory, the mat ID and the mat address. The mat ID can either be represented as a bit vector or a mat ID number and mat ID mask. This latter representation restricts multicasting to groups of powers-of-two mats along powers-of-two boundaries. The mat address is simply the internal mat address of the requested word.

Because the reconfigurable memory system takes the place of the hardwired first-level memory system of the processor, we need low latency access to avoid increasing the delay of critical pipeline loops. Thus, we choose to employ a hardware translation method similar to the address centrifuge used in the Cray T3E [92] to perform the virtual-to-hardware address translation. For each of the fields we must generate, we extract a portion of the virtual address and use it as an offset from a base value. The base value is either statically assigned to the issuing port or determined from the high-order bits of the virtual address. While this type of hardware address translator reduces the addressing flexibility, it is fast, avoiding large table look-ups.

The *address splitter* unit performs the virtual-to-hardware address translation. The mat ID base, mat ID mask, and mat address base are determined from a lookup table index by the high-order bits of the virtual address. We call these high-order bits the *logical memory ID*. They identify which logical memory structure to access. A logical memory is a collection of mats that make up a logical memory unit, such as a cache tag array, cache data array, scratchpad memory, or FIFO. The logical memory ID also determines the fields of the virtual address that are extracted to form the mat ID offset and mat address offset. The logical memory ID can have different interpretations depending on the processor port. So the same virtual address sent to processor ports handling a cache's tag and data, will route the requests to different mats, the tag mats and data mats respectively.

The extractor unit extracts the mat ID offset and mat address offset from the virtual address. Each extraction is a mask and shift operation. These offset fields are added to

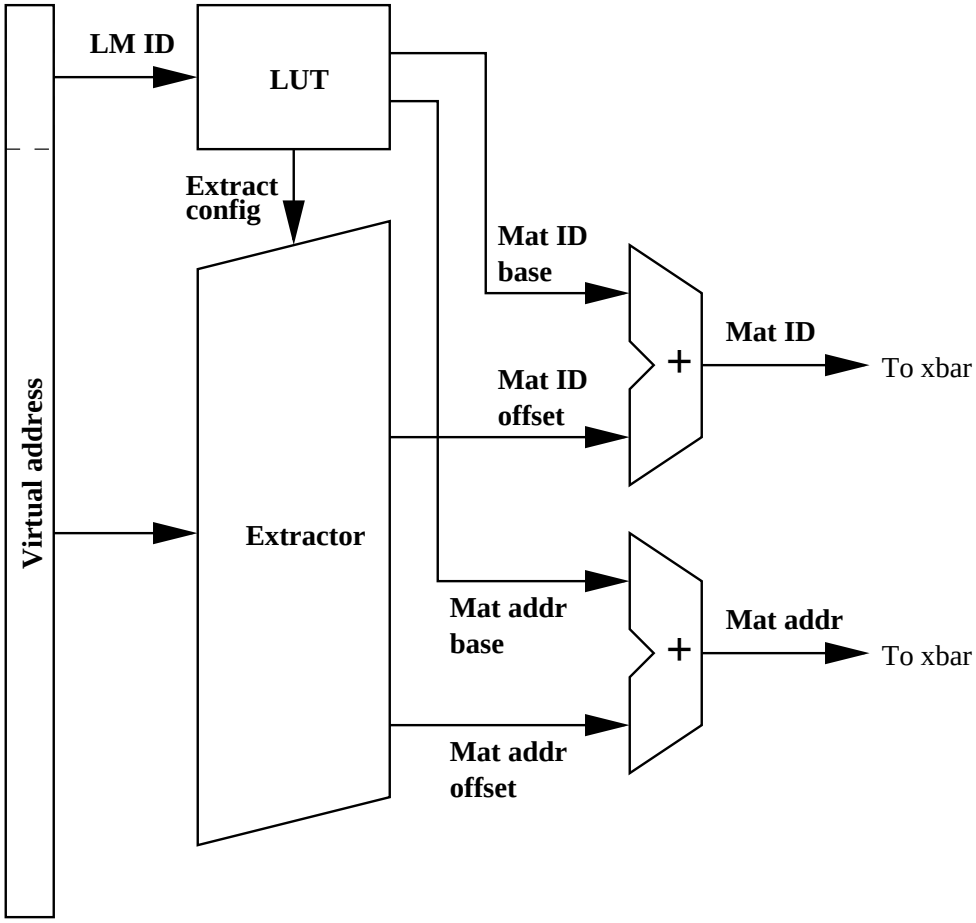

Figure 4.14: Address splitter block diagram

previously determined base fields to generate the final mat ID and mat address. Having a mat address base field allows a single mat to be shared among multiple logical memories. In accesses to cache tag arrays, the unused remaining portion of the virtual address is used as input data to the mat for the compare.

We can avoid added delay of the logical memory ID lookup tables if each processor port only ever accesses one logical memory. For example, if a processor port is dedicated to be the instruction cache tag port, then the mat ID base, mat ID mask, mat address base, and offset field split can be held constant, and no table lookup is necessary.

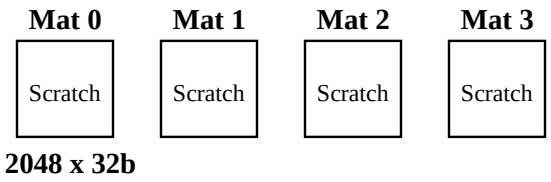

Figure 4.15: Scratchpad mat configuration for address splitter example

Figures 4.16 and 4.17 show example address splitter configurations for a scratchpad memory detailed in Figure 4.15. The example memory system has 16 mats each holding 2048 32b word, and the scratchpad memory spans four mats, mats 0 to 3. The address split in Figure 4.16 is for a configuration with the scratchpad words packed contiguously into four mats. The address split in Figure 4.17 is for a configuration with the scratchpad words interleaved among the four mats. So word 0 is in mat 0, word 1 in mat 1, word 2 in mat 2, and so on. This banking arrangement would be desirable to avoid request conflicts if there were multiple requesters for the scratchpad. In a statically schedule crossbar, this would allow multiple requesters to simultaneously access the scratchpad as long as their requests were guaranteed to access different banks.

The address split for a cache is more complex. Figure 4.18 shows an example 2-way set associative cache. Each way's tag is held in one mat, and four mats make up each data array. Each way holds 2048 4 word lines. Mats 0 and 1 are the two tag mats, and mats 8-15 make up the two data arrays. This mat numbering allows for easy multicasting of the tag and data accesses, since the tag and data arrays are power-of-2 in number and along

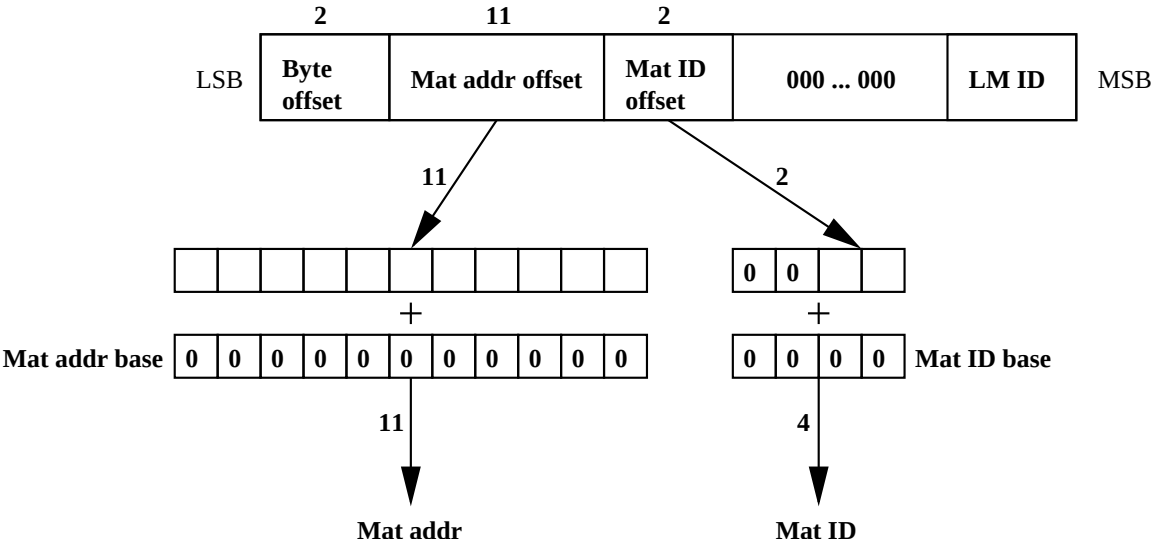

Figure 4.16: Address split for contiguous word scratchpad

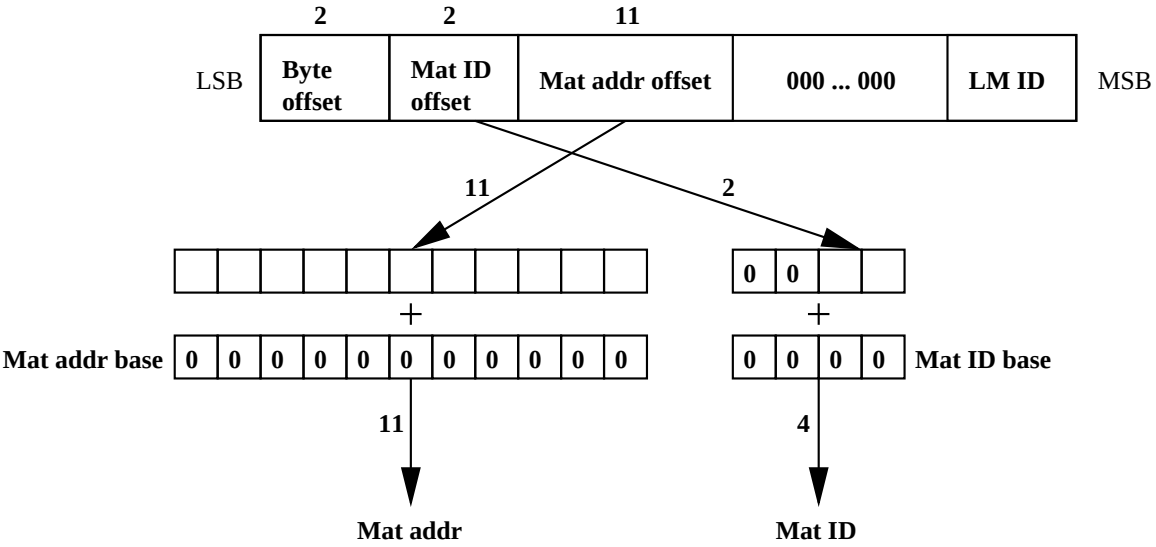

Figure 4.17: Address split for interleaved word scratchpad

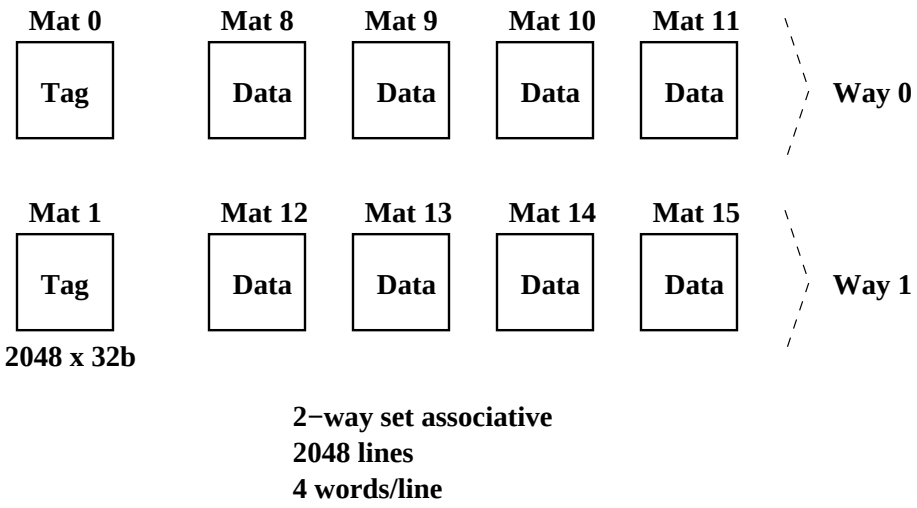

Figure 4.18: Cache mat configuration for address splitter example

power-of-2 boundaries.

Figures 4.19 and 4.20 show the address splits of the tag and data requests respectively. The data split assumes that the lines are packed contiguously into the data mats. Note that in the tag array, the remainder portion of the virtual address is used as compare data in the tag access. Figures 4.21 and 4.22 show the address split again for the tag and data, but for a cache with the line interleaved among the four data array mats.

The address splitter logic could be combined with the regular address generation logic typically done in the EX processor pipeline stage. Additionally, the mat decoder could be implemented a sum-addressed-decoder [139] to remove some of the logic from the address splitter. This would however increase the width of the address field sent to the mat. We assume that the traditional virtual-to-physical address translation occurs at higher levels of the memory hierarchy. The local caches built from mats are virtually-indexed, virtually-tagged.

To make up a traditional memory request (*i.e.* full data-width address and data) to a cache would require two ports, one for the address and one for the data. The ”address” of the request packet is just the hardware address that the payload is sent to. For example, in a system with a cache, the physical address is really just the data portion of one of the ports which is sent to the mat(s) acting as the cache tag.

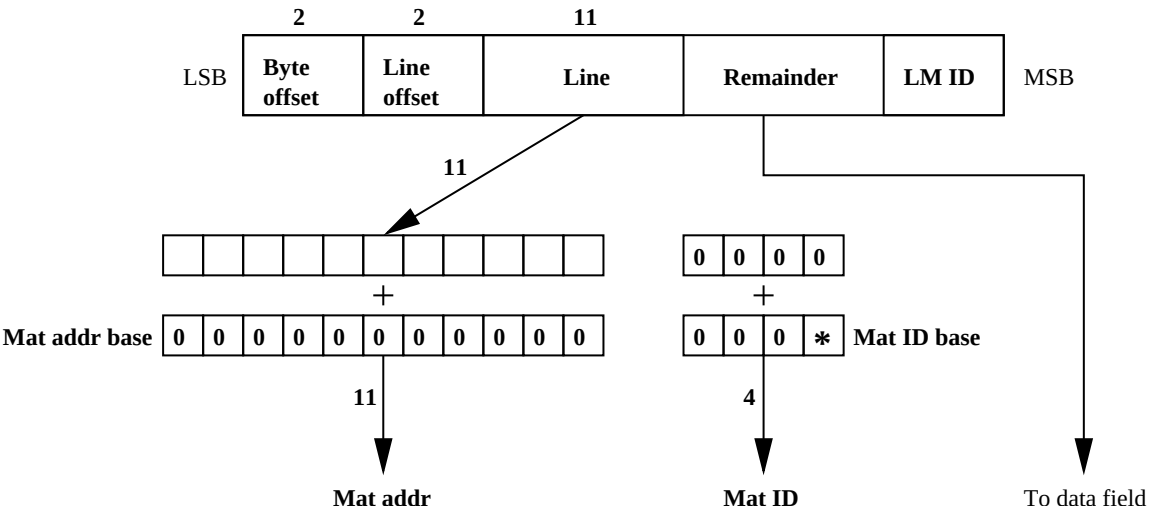

Figure 4.19: Cache tag address split for contiguous word data array

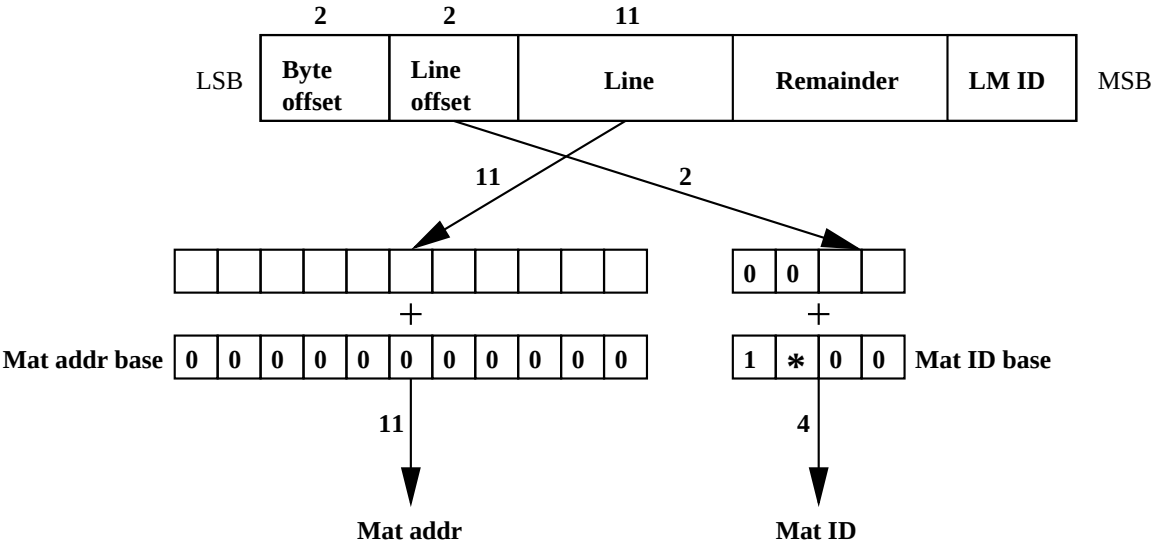

Figure 4.20: Cache data address split for contiguous word data array

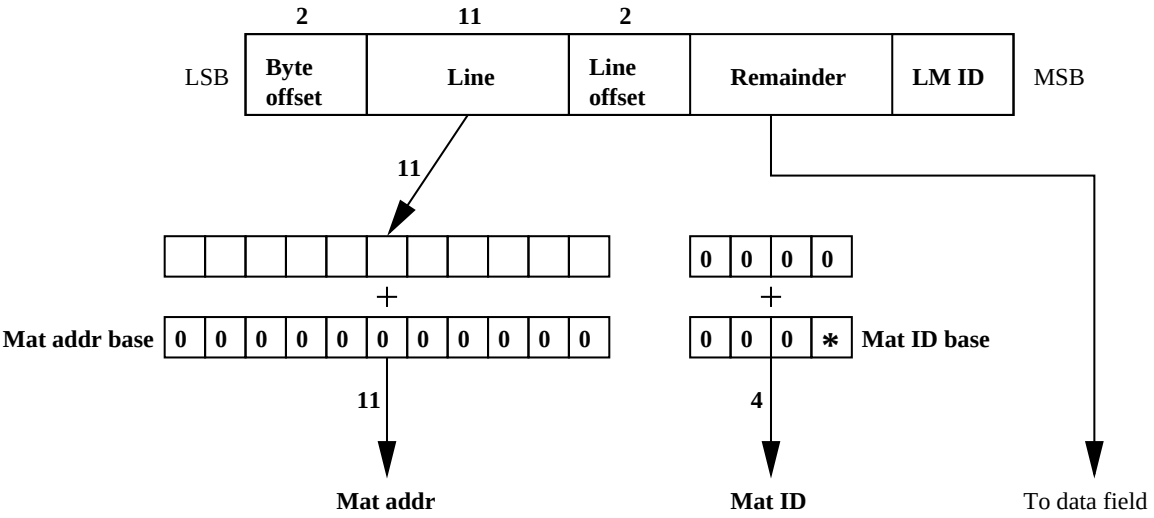

Figure 4.21: Cache tag address split for interleaved word data array

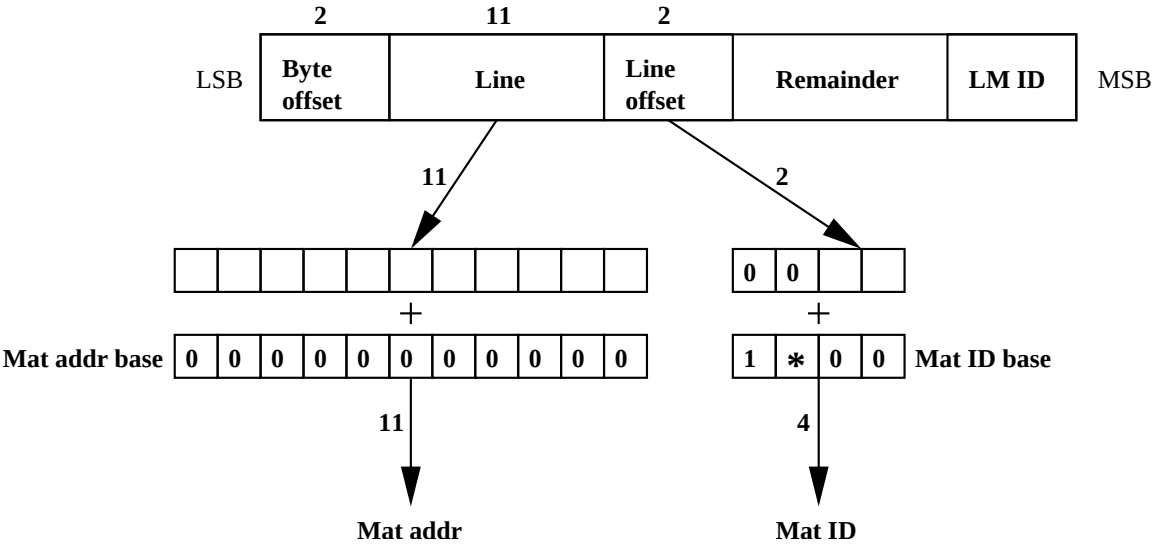

Figure 4.22: Cache data address split for interleaved word data array

If a system had 8 ports, the ports could be configured as 4 traditional memory ports. However, if the processor was only going to address local memory, then all 8 ports could be used to push/pull a data word every cycle. Any mix of the above is also possible. A single address port could be associated with multiple data ports to pull down a wide data word. Or some ports could use a traditional address space scheme, while others directly accessed the hardware addresses.

4.4 Summary

This chapter described the two interconnection networks in our reconfigurable memory system: the inter-mat control network and the processor interconnect network. The inter-mat control network allows the mats to pass a few bits of control information to each other. It uses narrow, one-to-many communication, and we implement it as a number of segmented buses. The processor communicates to the mats via the processor interconnect network. It must support wide, many-to-many communication with many concurrent requests. We implement this network as two uni-directional crossbars, one for the requests from the processor ports to the mats and one for the replies from the mats back to the processor ports. There are many ways to implement these networks, and we chose interconnect topologies that were suited to their respective communication patterns and emphasized flexibility over area or energy efficiency. But the optimal topology depends heavily on the specific memory architecture and optimization goals.

Chapter 5

Experimental Results

In the previous chapters, we have detailed a reconfigurable memory design based on a reconfigurable SRAM mat and a flexible interconnection network. Our design goal was to make the memory flexible, yet maintain high performance and efficiency. As a proof-of-concept for our design, we implemented a prototype reconfigurable memory testchip. The testchip demonstrates that we can build a fast, flexible, efficient reconfigurable memory, sanity checks our design decisions, and quantifies the reconfigurability overheads. From the prototype results, we can also extrapolate the overheads for larger, more complex reconfigurable memory designs.

5.1 Testchip Overview

We designed and fabricated a reconfigurable memory testchip in a 0.18μm CMOS 6-metal Al TSMC process (Figure 5.1) [140][141]. The die measures 3mm by 3.3mm and has 68 pads along the periphery. It was packaged in a 84-pin ceramic leadless chip carrier. Table 5.1 summarizes the testchip and process technology features.

Figure 5.2 shows the testchip block diagram. The testchip contains four reconfigurable memory blocks (Mem0-3), a dynamically routed, low-swing crossbar interconnect, test vector storage, and a process monitor block. Mem0 contains an SRAM core that uses a self-timed, pulsed-mode circuit style for fast access and short cycle time. Mem1 and Mem2 contain complete memory mats using the SRAM core from Mem0. Mem3 contains

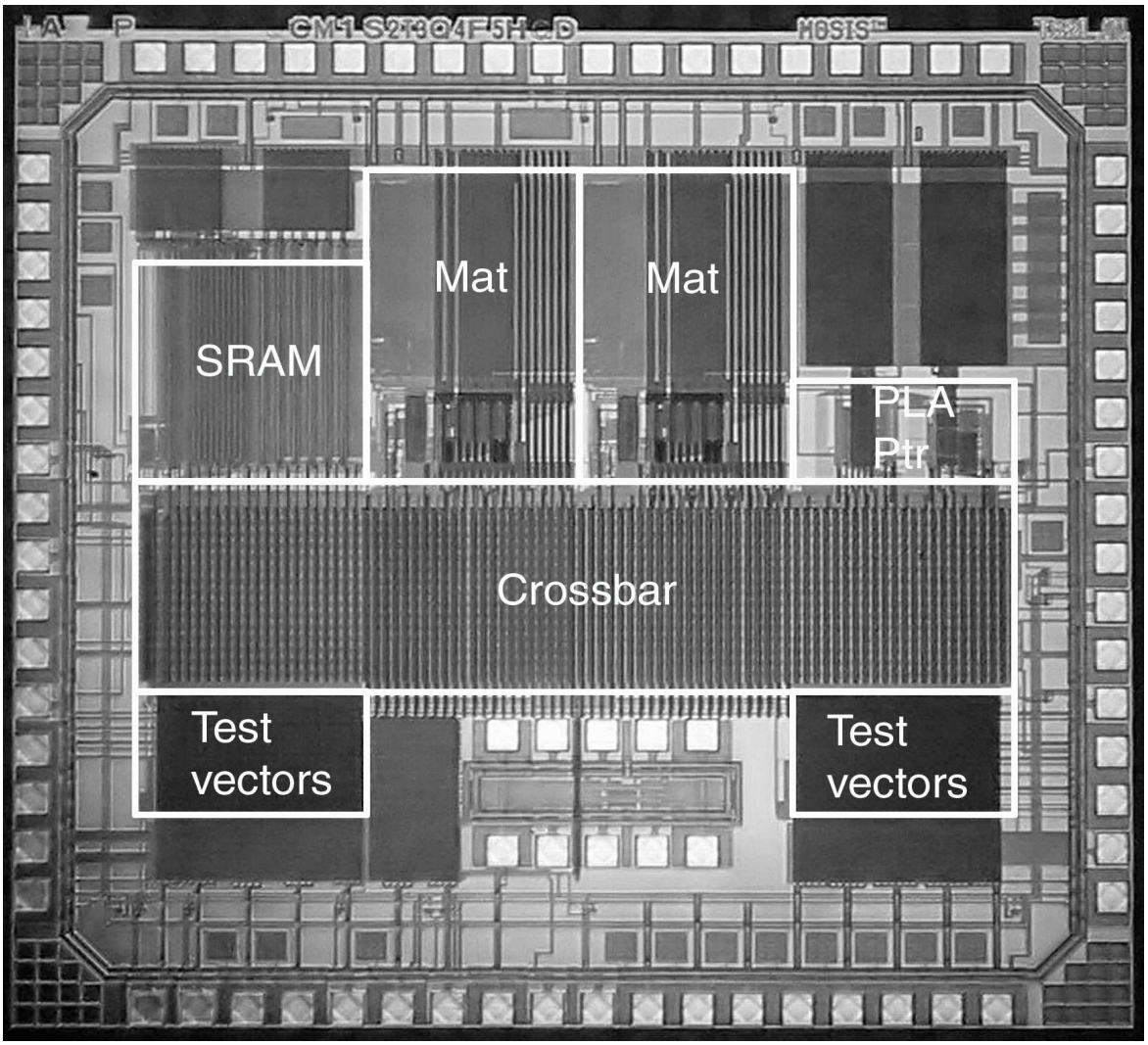

Figure 5.1: Testchip die photo

Table 5.1: Process and testchip features

Mat area	0.6 mm^2
SRAM area	0.4 mm^2
Cell area	9 μm^2
Meta-data cell area	18 μm^2
Supply voltage	1.8V
Frequency	1.1GHz
Mat power dissipation	125 mW (50% reads, 50% writes)
Technology	0.18μm CMOS, 6-metal Al
Transistors	NMOS Vt = 0.5V, PMOS Vt = -0.5V

the pointer logic and PLA used in the mats. We separated out the Mem0 SRAM and the Mem3 peripheral blocks for increased observability and to isolate their power measurements. Mem0 and Mem3 use isolated power supplies so that we can measure the power dissipation of each block.

We chose an aggressive cycle time goal of 10 fan-out-of-four inverter delays (FO4) for the testchip. This short clock tick stresses the circuit design of the memory cores as well as the peripheral circuits and interconnect. Also, this cycle time would allow us to virtually multi-port the memory system for a slower cycling processor, as described in Chapter 3. Due to this aggressive cycle time, the mat access is pipelined, and the total delay through the mat is 2 cycles or 20 FO4. The first half-cycle is spent in the pre-access logic: pointer logic or write buffer. The next full cycle is spent in the SRAM access. The last half-cycle is spent in the post-access logic: control logic and the comparator. The mat is fully pipelined, accepting a new request every 10 FO4 cycle. For each crossbar traversal, we allocate a full 10 FO4 cycle. An entire memory access requires 4 cycles, or 40 FO4.

5.1.1 SRAM core

The SRAM core used in Mem0, Mem1, and Mem2 has a capacity of 18Kb. This is on the small end of the 16Kb to 128Kb optimum energy-delay range discussed in Chapter 2. We chose this capacity to keep the area of the testchip reasonable, while still being able to include a number of SRAM cores on the die. The SRAM holds 16Kb of data, arranged as

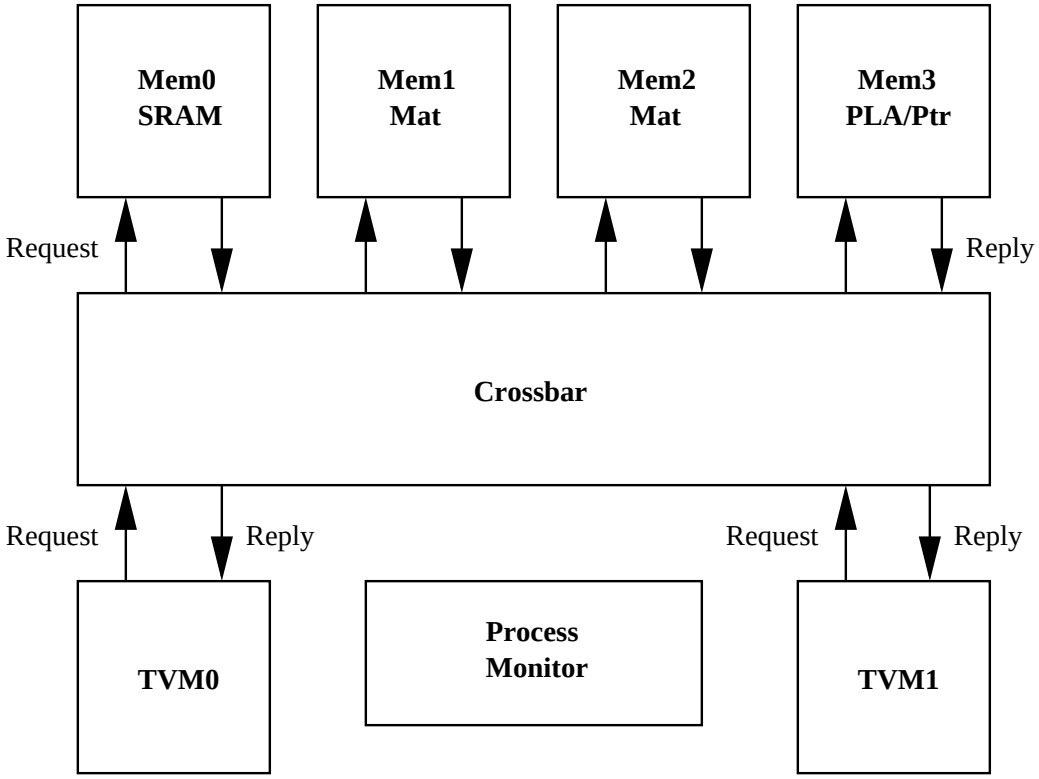

Figure 5.2: Testchip block diagram

512 32b words. Along with each data word, we store an additional 4 bits of meta-data, so the SRAM is logically 512 x 36 bits. To balance the wordline and bitline lengths, we use 4:1 column multiplexing to achieve a nearly square cell array of 128 x 144 cells. The row address is then 7 bits and the column address is 2 bits.

Our decoder structure closely follows the optimal topology presented by Amrutur and Horowitz in 2001[34]. The pre-decoder divides the 7 bit row address into two groups and decodes them using a 3:8 decoder and a 4:16 decoder. Each of the 24 pre-decode gates uses a modified Nambu OR-gate (Figure 5.3) [36][142]. We insert an always-on full CMOS transmission gate in the clock path to the dynamic inverter to slightly delay its clock. This reduces the output glitch on the gates that do not fire.

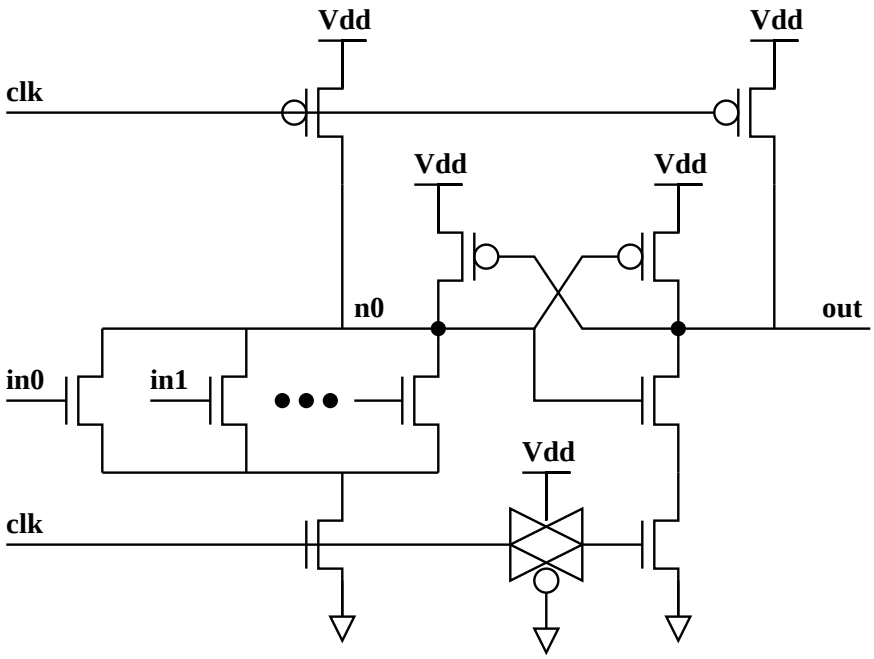

Figure 5.3: SRAM predecoder gate

The wordline driver gates are self-resetting, two-input, source-driven AND gates (Figure 5.4) [37][38][33][40]. There is an explicit pull-down cut-off device to avoid drive fights during reset. The 3:8 pre-decoder generates the negative pulses *in_n[7:0]* that drive the source inputs of the wordline driver gates. The 4:16 pre-decoder generates the positive pulses *in_p[15:0]* that drive the gate inputs of the wordline driver gates. The buffer delays

for the 3:8 pre-decoder and 4:16 pre-decoder are designed to be matched so that the positive and negative pulses arrive at the wordline driver gates at the same time.

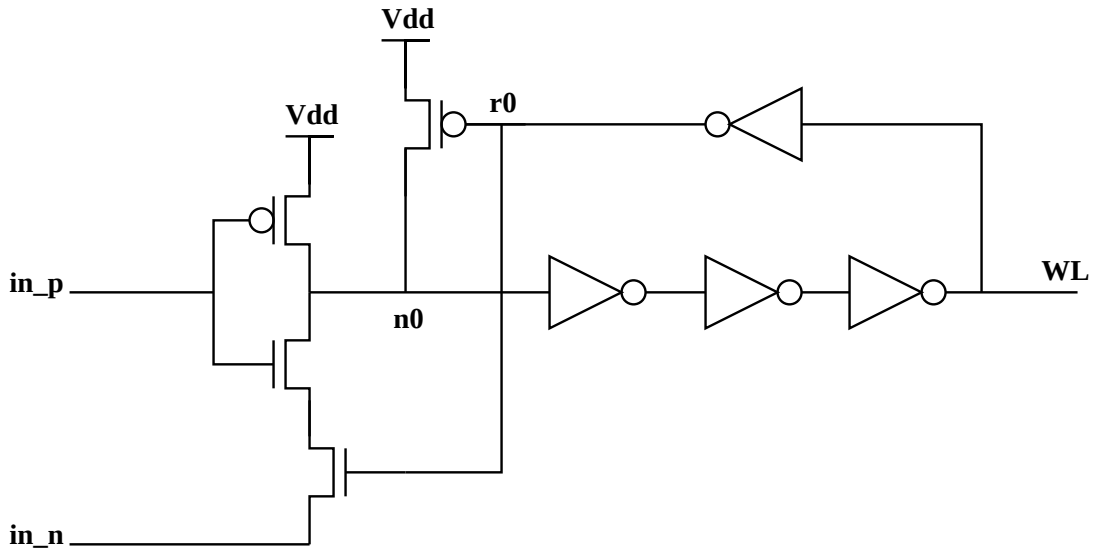

Figure 5.4: SRAM wordline driver gate

The decoder also contains the read-modify-write decoder as described in Chapter 3. The decoder drives the wordlines into the cell array. The cell array contains both the two-ported meta-data cells and the single ported data cells. The meta-data cells fit in the cell pitch of the regular single-ported data cells, but they are wider to accommodate the two pairs of bitlines. The prototype meta-data cells support read-modify-writes and gang operations, but not conditional or conditional gang operations.

The cell bitlines feed into the cell I/O block that contains the read and write support circuits. The read cell I/O consists of a 4:1 PMOS passgate column mux and a StrongARM amplifier followed by a skewed SR latch (Figure 5.5) [57]. We chose this design over the more common latch-based sense amplifier to remove a series device from the bitline data path. To further reduce the bitline loading, the write driver is pull-down only which allows the write column mux to be a simple 4:1 NMOS passgate mux.

After an access, the bitlines must be reset to Vdd before the next operation. The bitline reset circuits (Figure 5.6) consist of a keeper (*M0* and *M1*), read reset (*M2*, *M3*, and *M4*), and write reset (*M5* and *M6*). During writes the bitlines swing full-rail, and we need large

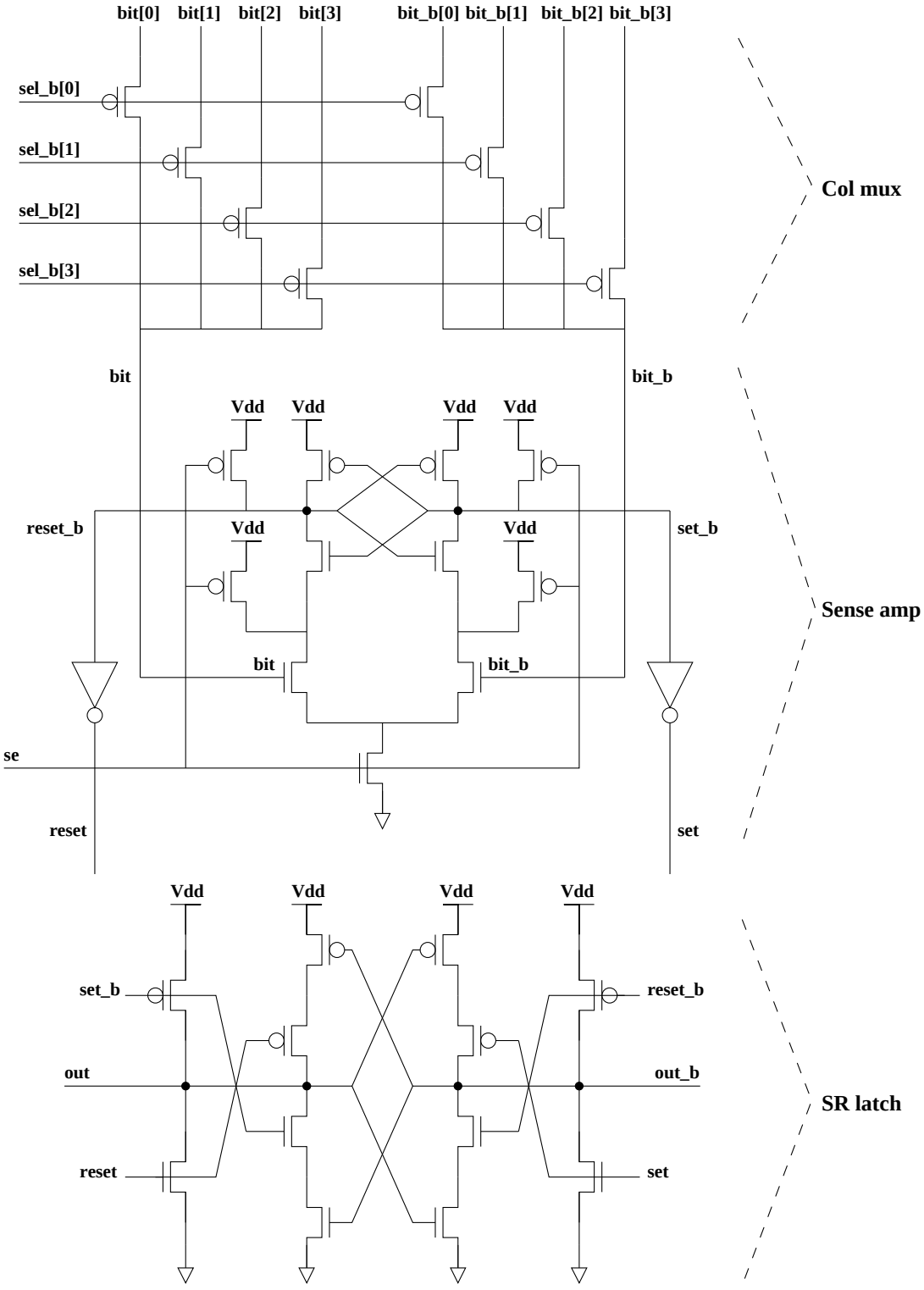

Figure 5.5: SRAM read I/O circuits

devices to fully reset the bitlines before the next access. However, only 1 out of 8 bitlines needs to be reset from Gnd to Vdd, because we use 4:1 column multiplexing and only one bitline per activated pair is discharged in a write. We use self-resetting bitlines to avoid the excess power dissipation of activating unnecessary write reset devices. A self-reset delay chain activates the write reset devices, *M5* and *M6*. The self-reset activation is predicated on write enable being de-asserted (*wr_en_b* being high) to prevent a drive fight between the reset device and the write driver.

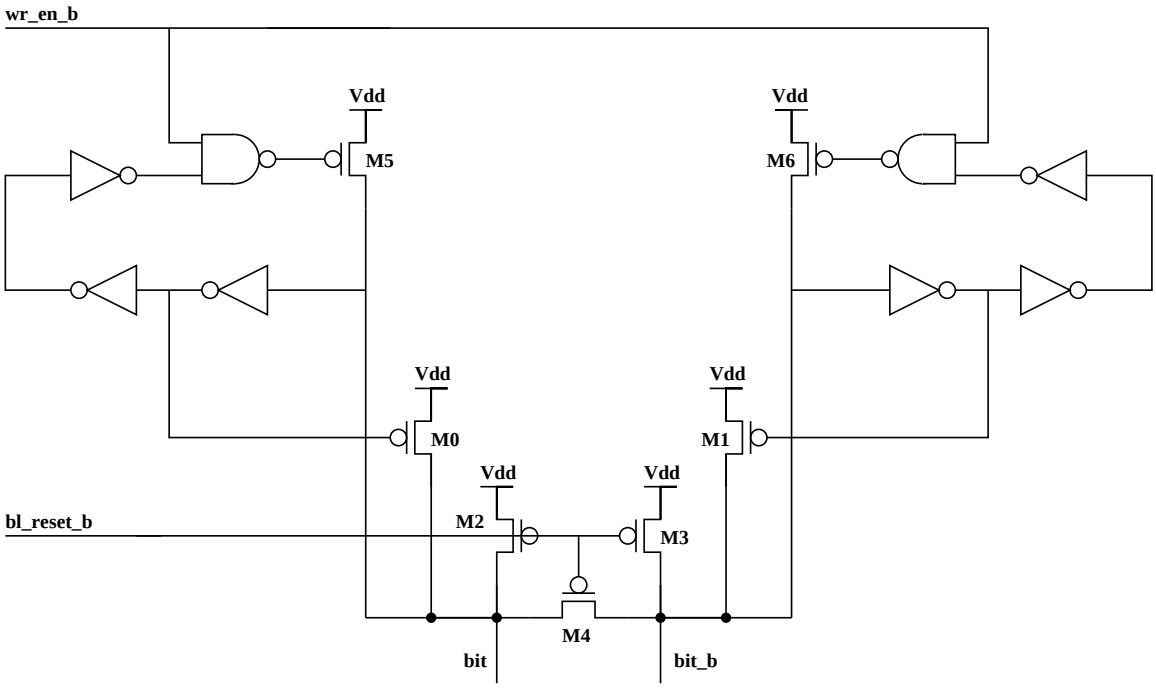

Figure 5.6: SRAM bitline reset circuits

We use a current ratioed replica bitline [35] to time the activation of the clocked sense amplifiers, for good tracking of the replica path with the actual data path across process, voltage, and temperature variations. The full replica timing path consists of a mimic pre-decoder gate, mimic wordline driver gate, replica wordline, scan tunable replica cell, replica bitline, and sense enable buffer. The delay of the sense enable buffer is matched to the delay of the pre-decoder output driver using logical effort [32]. Figure 5.7 shows both timing paths and the matching of each of the delays such that the sense enable signal is

aligned in time with the desired bitline split of 100mV.

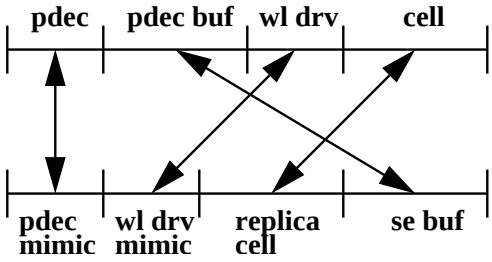

Figure 5.7: SRAM replica timing path

Proper replica wordline loading requires an extra row of dummy cells, and proper replica bitline loading requires an extra column of dummy cells. Despite the careful matching, the replica path delay can be slightly different from the data path due to the slow edge rate of the replica bitline, device mismatch, or inexact delay tracking across PVT variations. To combat the possible problems with replica path matching, we implemented a tunable replica bitline pull-down. Ten replica bitline drivers, two per driver cell, are laid out in the replica row as shown in Figure 5.8. A scan-set-able register determines the number of drivers enabled via the *drive[n]* signals. In simulation, the replica path matched the data path at the middle setting with 5 drivers enabled. This allows for 5 steps in either direction, faster or slower, if the replica path does not match. The prototype implementation performed at its maximum 1.1GHz clock frequency with the replica bitline driver at the nominal setting.

5.1.2 Peripheral Logic

Mem1 and 2 are complete memory mats containing the SRAM core, reconfigurable PLA, pointer logic, maskable comparator, write buffer, and control logic. The reconfigurable PLA has 16 logic terms (*i.e.* rows) with 6 input and 4 outputs. The 6 inputs are 4 bits of meta-data, the 1 bit match result from the mat comparator, and one external bit. The 4 outputs correspond to the 4 meta-data bits.

The pointer logic stores pointers that are 11 bits long, which is 2 bits longer than necessary to address all 512 words in the main SRAM array. The 2 extra bits allow a maximum

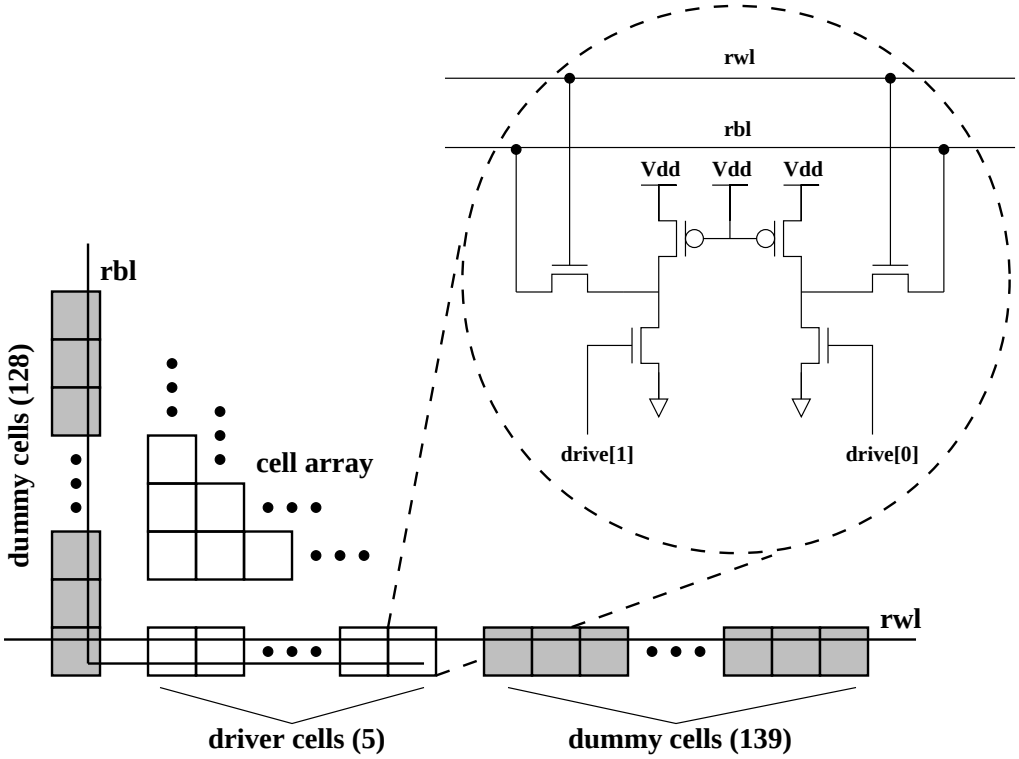

Figure 5.8: SRAM replica bitline and wordline

FIFO size of 4 mats. The pointer storage array holds 4 pointers allowing for a maximum of two FIFOs per mat. The strides were chosen to be 4 bits long. The pointer storage array decoder gate is a self-resetting, pulse-latch with embedded logic (Figure 5.9). We implement the adder/subtracter using complex dual-rail domino gates and uses a mixed carry-tree/carry-select topology. We previously detailed the pointer logic micro-architecture in Section 3.6.2.

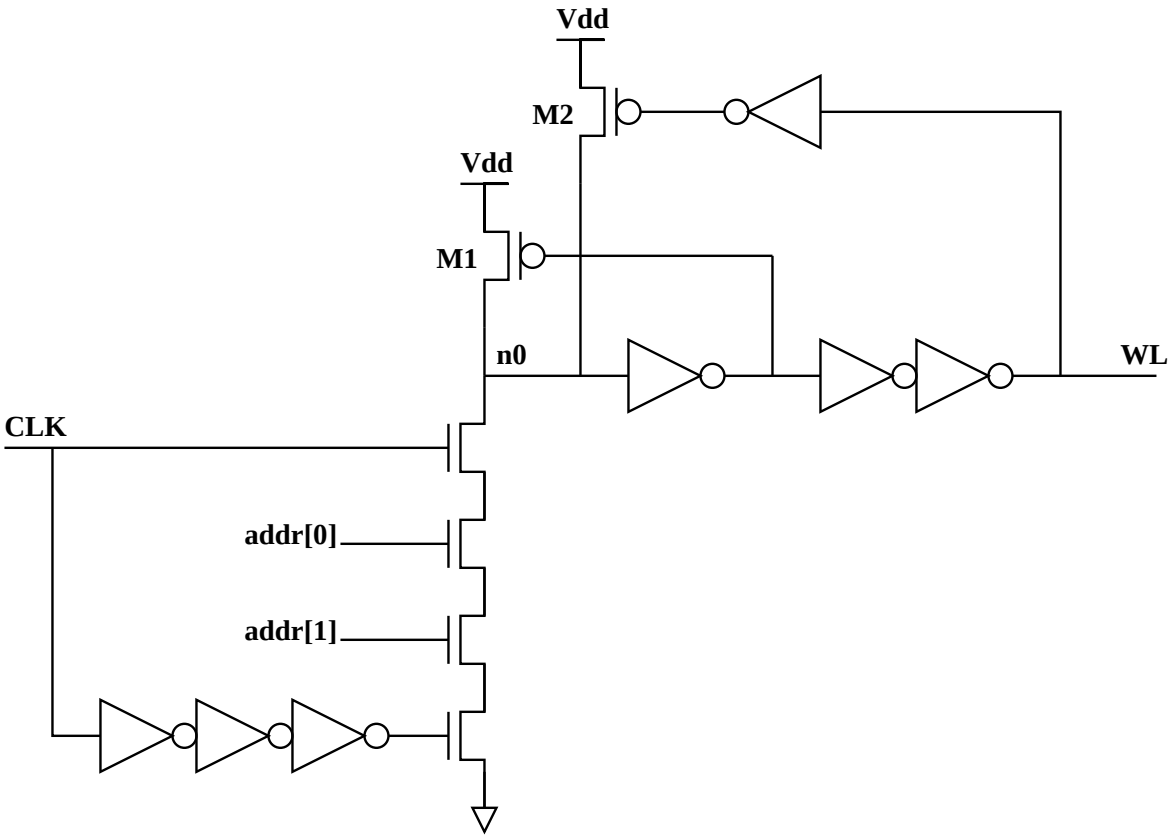

Figure 5.9: Pointer logic decoder gate

5.1.3 Interconnect and Test Infrastructure

The memory blocks connect to the two test vector memories via two dynamically routed, uni-directional, low-swing crossbars. The request crossbar routes 60b request packets from

the two test vector memories to the four memory blocks. It supports multicasting to any combination of the memory blocks. The reply crossbar routes 40b reply packets from the memories back to the test vector memories. It supports muxing of the return data as detailed in Chapter 4. We used low-swing wires in the crossbars in order to reduced their power dissipation. If we had used full-swing wires, the crossbars would have burned as much power as the rest of the testchip. We implemented low-swing differential signaling using NMOS-only drivers and clocked sense amplifiers as described in Chapter 4.

The two test vector memories (TVMs) emulate processor ports sending requests into the reconfigurable memory system and receiving the replies. Each TVM stores 16 64b requests and can launch a request every cycle. The TVMs also capture the replies coming back from the memory system. Each TVM can store 16 40b replies, accepting one per cycle. Putting the TVMs on die allows us to test the memory mats at speed without using high-speed I/O pins or a high-speed tester.

We implement the TVMs as wide looping shift registers that output a memory request and record a response every cycle. All of the test vector memory storage cells are connected in a serpentine scan chain which allows us to read and write the input and output vectors via a simple, low-speed, scan interface. The TVMs can be run in a looping mode where the 16 requests are repeated over and over again. This mode is useful for taking power measurements and for probing internal nodes with the on-die voltage samplers.

The testchip has 43 voltage samplers on key internal nodes and the clock. These voltage samplers were described by Ho in 1998 [143], and our sampler design is identical to that used on an earlier testchip in the same technology [56]. The sampler uses a boosted supply voltage to allow the NMOS sample-and-hold passgates to sample voltages up to the normal core Vdd. By running the sampler clock at a slightly different frequency as the core clock, we sub-sample the signal and can generate a time dilated version of the signal. This time dilated signal is a much lower frequency than the actual signal and is easily driven off chip. By using the clock sampler output as a reference, we can relate the sampler time-dilated output to actual time. In the next section, we show a waveform capture of a number of key nodes in the SRAM core using the samplers.

The testchip contained a process monitor block (Figure 5.11) which allows us to measure the delay of various logic gates including the fanout-of-four inverter delay of each die

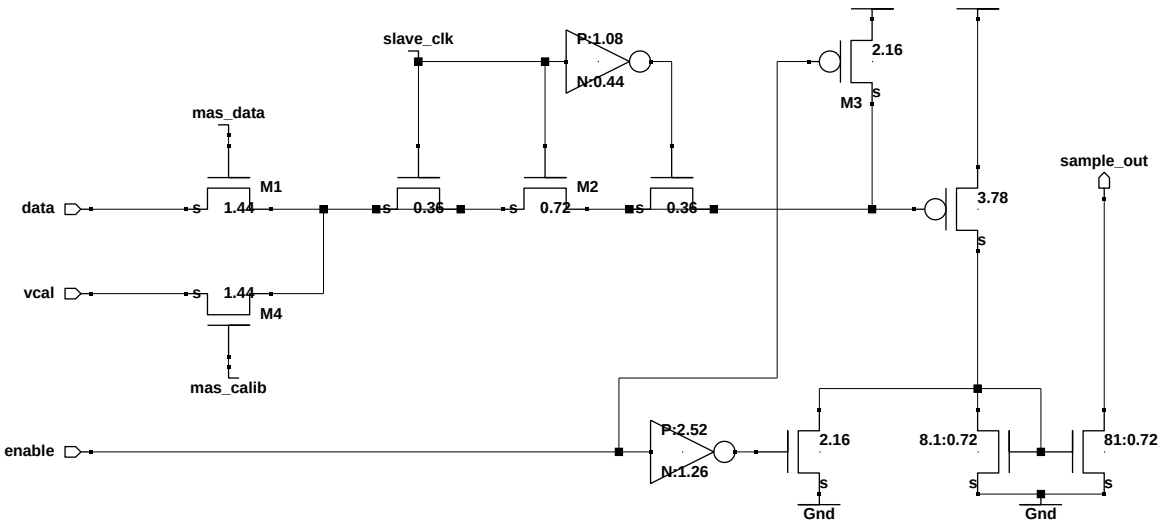

Figure 5.10: On-die voltage sampler with device widths in μms for a 0.18μm technology[56]

[144]. The block contains four ring oscillators, each using a different type of gate for the delay element. A mux selects which of the ring oscillator outputs goes to a by-32 toggle flop frequency divider. Its output is buffered then sent to an output pad. The block has 10 dedicated power and signal I/O pads. The pads are not along the periphery of the die and require special bonding.

5.2 Measured Results

The testchip operated at 1.1GHz at the nominal 1.8V supply and room temperature. The process monitor block measured the fanout-of-four inverter delay to be an average of 89ps across 5 die. The 1.1GHz operating frequency corresponds to a target cycle time of 10 FO4. The area breakdown for the SRAM core, memory mat, and testchip are shown in Tables 5.2, 5.3, 5.4 respectively. The meta-data, meta-data cell I/O, and the RMW decoder occupy 24% of the total SRAM area.

Figure 5.12 shows the percentage area breakdown of the mat. The peripheral logic occupied 32% of the mat area, but over half of the 32% was routing area. With better optimized layout, we could have reduced the routing area significantly. Figure 5.13 shows

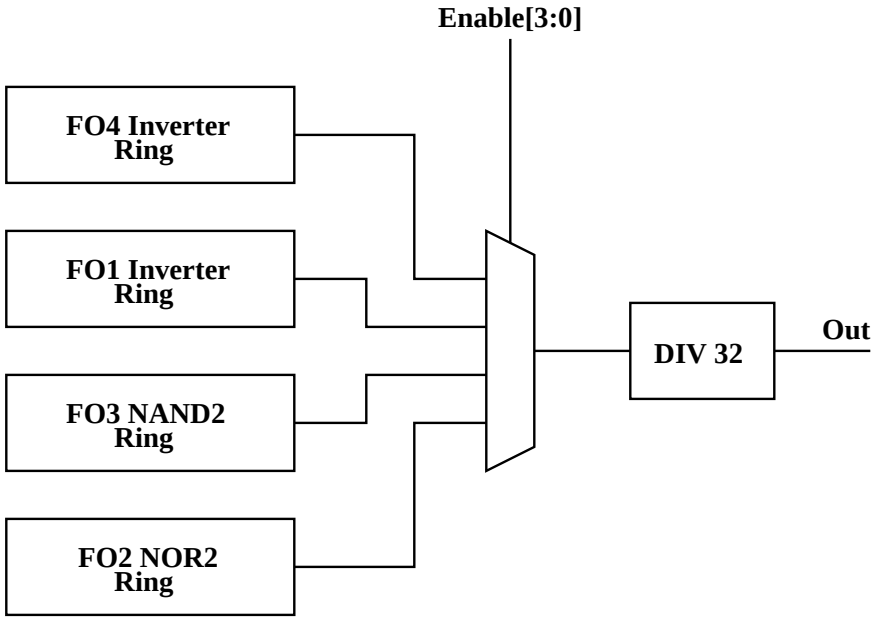

Figure 5.11: Process monitor block

Table 5.2: SRAM area breakdown

Unit	Area mm^2
Decoder	0.029
RMW decoder	0.029
Mdata array	0.052
Data array	0.186
Data cell I/O	0.036
Mdata cell I/O	0.012

Table 5.3: Mat area breakdown

Unit	Area mm^2
SRAM	0.376
PLA	0.021
Pointer	0.017
WB/Cmp	0.05
Routing	0.097

Table 5.4: Testchip area breakdown

Unit	Area mm^2
SRAM	0.376
Mat	0.637
PLA/Ptr	0.047
Crossbars	1.62
TVM	0.241
Process Monitor	0.388

Table 5.5: Mat power breakdown

Unit	Power mW
SRAM	104.4
PLA	9.9
Pointer	6.3
WB/Cmp	4.4

the area overhead as a function of the mat capacity, keeping the data and meta-data widths constant. The projected area overheads are generated using an SRAM area estimator [24] and and the testchip area results. With a 64Kb mat, the peripheral logic occupies less than 15% of the total area, still using the non-optimal peripheral logic layout.

The power breakdown for a memory mat is shown in Figure 5.14 and Table 5.5. The peripheral logic accounted for 23% of the power. The test vector was an even mix of compare-modify-writes and pointer writes. The PLA function was a 4-bit counter. Figure 5.15 shows the power overhead as a function of the mat capacity. As with the area projections, the data and meta-data widths are kept constant. The projected power overheads are generated using an SRAM power estimator [24] and and the testchip power results. With a 64Kb mat, the peripheral logic accounts for less than 10% of the power.

Figure 5.16 shows a waveform capture from the SRAM using the on-die voltage samplers for a worst-case read after write at 1.0GHz and 1.8V. The write occurs from time -1ns to 0ns, and the read occurs from time 0ns to 1ns. The bitlines have approximately 90mV

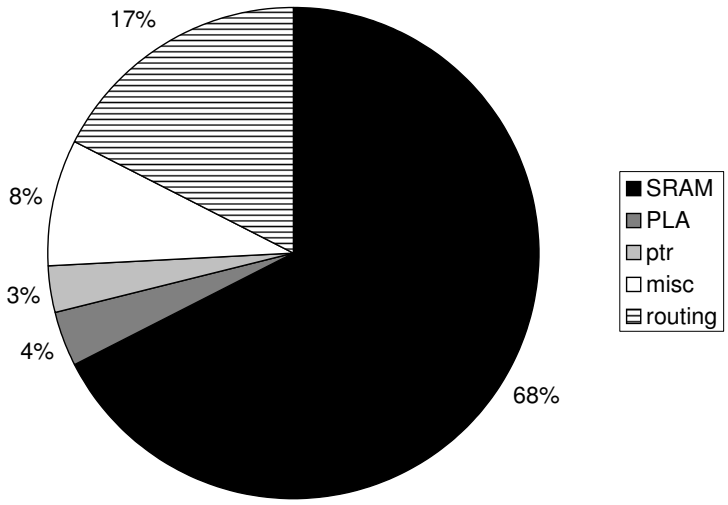

Figure 5.12: Area overhead and breakdown

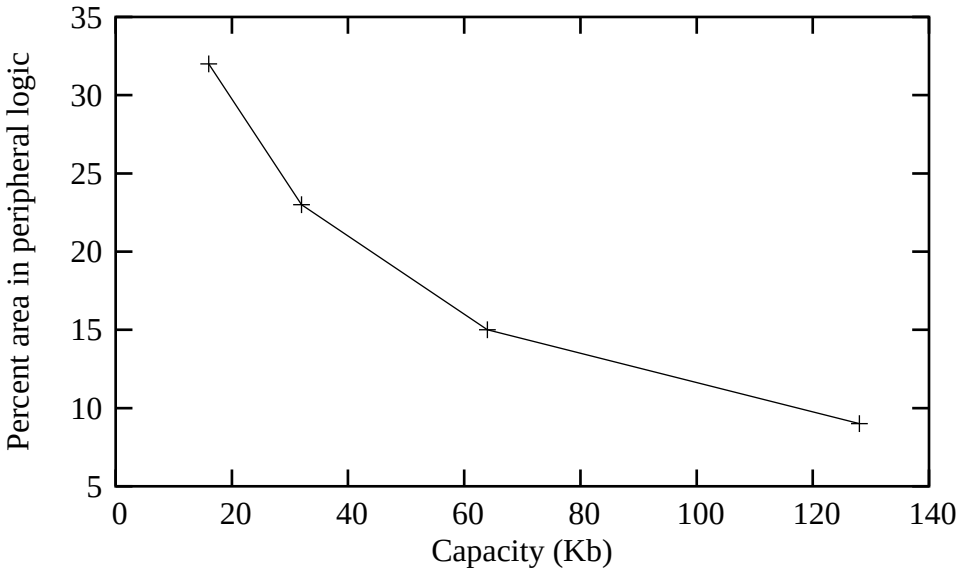

Figure 5.13: Area overhead vs. SRAM capacity

of split when the sense enable signal fires. Due to insufficient decoupling capacitance on the sampler supply, the bitline samples are DC shifted up by about 200mV. The jaggedness of the bitlines near Gnd is due to noise and increased sensitivity of the sampler calibration near ground.

5.3 Summary

This chapter described the implementation of a prototype reconfigurable memory testchip and the measured results. The testchip contained four memory test structures, including two complete 16Kb memory mats, a low-swing crossbar interconnect network, test vector storage, and a process monitor block. The testchip operated at the target 10 FO4 clock cycle under nominal conditions. The fast cycle time of both the memory mats and interconnect opens the possibility of virtual multi-porting the memory system. The reconfigurable logic occupied 32% of the mat area and accounted for 23% of the mat power. The mat memory capacity is on the small end of the optimal range, and we project that for larger memory

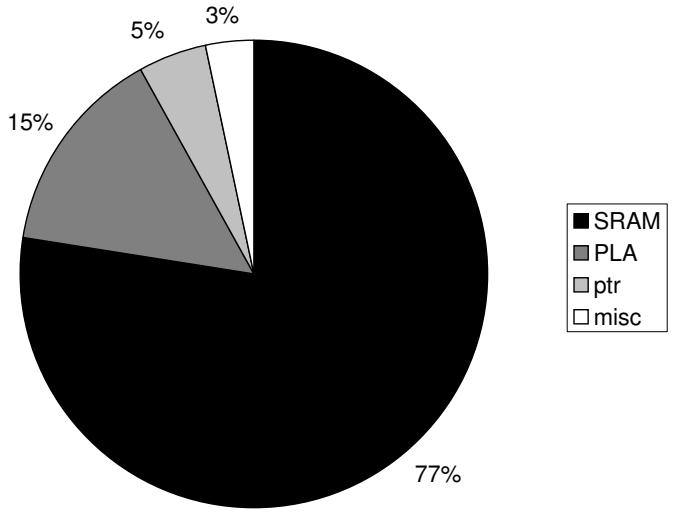

Figure 5.14: Power overhead and breakdown

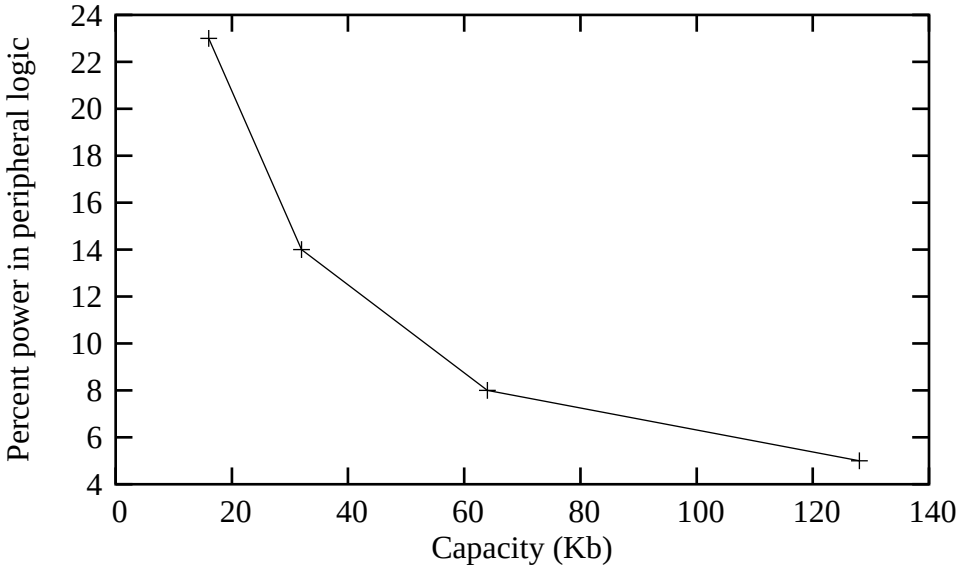

Figure 5.15: Power overhead vs. SRAM capacity

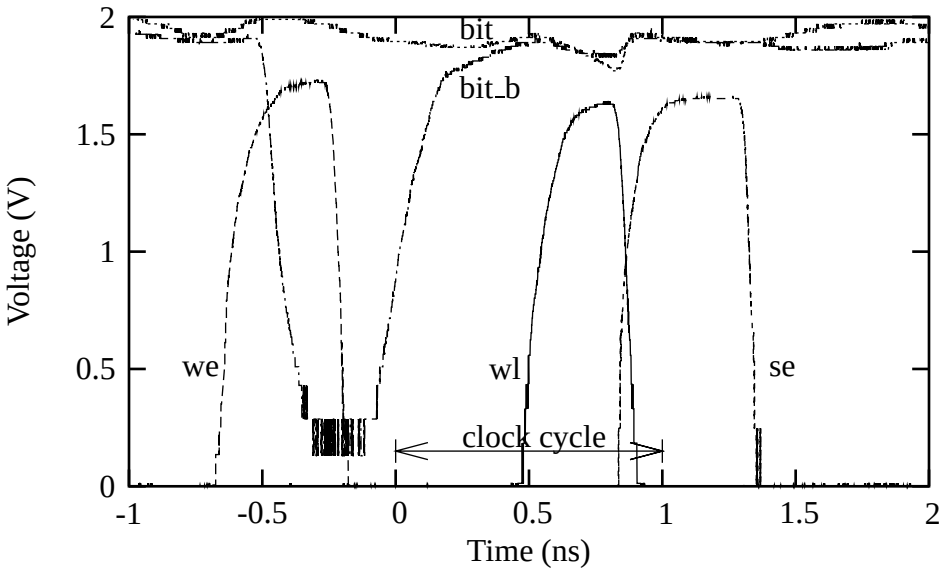

Figure 5.16: Worst-case read at 1GHz, 1.8V, room temperature

capacity mats, the overheads for area and power would fall below 15% and 10% respectively. These results show that we can implement the proposed reconfigurable memory design with reasonable overheads while maintaining high-performance operation.

Chapter 6

Conclusions

In this work, we have examined a reconfigurable memory architecture and evaluated its feasibility via a prototype testchip. The motivation for such an architecture stems from current process technology and design trends that indicate that custom ASICs will become increasingly difficult and costly to design. However, future applications will require more efficient, high-performance computation than current general purpose processors can provide. One promising approach to breaking this impasse is to use reconfigurable architectures that keep the low non-recurring engineering costs of general purpose silicon, yet still provide the efficiency and high-performance near that of custom ASICs. For such reconfigurable architectures, we propose adding reconfigurability to the memory system as well as the computation. While reconfigurable logic has been studied extensively, the design space for reconfigurable memory is relatively uncharted. However, in modern designs, the memory system plays an increasingly important role in the overall performance, area, and power of the design.

Unlike previous attempts at reconfigurable memory design, we took a bottom-up, circuit-level approach, starting with an efficient SRAM design and adding configurability where and when it could be done for low overhead. In Chapter 2, we explored modern SRAM design to understand the base design substrate. Modern SRAMs are highly partitioned designs that use specialized circuits in the decoder, datapath, and request/reply transport for high-performance and low-power. For the low-activity factor decoders, designers use self-resetting logic to achieve the speed of a dynamic logic family without the excessive power

dissipation of a global precharge signal. To save power in the bitlines, the read bitlines only swing a fraction of Vdd, but this requires a sense amplifier to restore the read out data to full logic levels. Clocked sense amplifiers are used for low power, but they require precise and robust triggering. Using replica bitlines allows us to accurately time the sense amplifier activation across process, voltage, and temperature variations. To reduce the power in the transport phases, we use low-swing signaling techniques on the wires. These highly optimized SRAMs are used to build common memory structures such as scratchpads, caches, and FIFOs.

Looking closely at these memory structures, we recognized that they use very similar memory building blocks. We designed a single reconfigurable memory mat that could form the core of a many common memory structures. To the memory array, we add a few extra bits of meta-data and logic to support read-modify-writes and gang operations. Pointer logic in the address path enables a level of address indirection for FIFO configurations. A comparator and write buffer in the mat datapath enable compares and conditional writes. For a realistic design evaluation, we use modern SRAM circuit design practices described in Chapter 2 in the mat design and tightly couple the meta-data and peripheral logic with the memory core.

Despite having very similar memory building blocks, the target memory structures vary widely in how they connect their memory blocks to each other and to the computation. To support this diversity, we use two flexible interconnection networks, the inter-mat control network (IMCN) for mat-to-mat communication and the processor interconnect network for mat-to-compute communication. These two networks have different communication needs and we tailor our implementations accordingly: the narrow one-to-many IMCN implemented as a segmented bus, and the wide many-to-many processor interconnect implemented as a pair of multicasting/muxing crossbars. To properly interface between the crossbar and the computation, the address splitter performs address translation from the address space used by the computation to the memory system hardware address space.

To evaluate our design, we implemented a prototype reconfigurable memory testchip based on our architecture in a 0.18μm CMOS technology. The testchip demonstrates that we can build such a reconfigurable memory with low overhead, while still maintaining a fast 10 FO4 cycle time. While this cycle time did require us to pipeline the memory

access, it opens the possibility using such a fast memory in a virtual multi-ported fashion, especially if the computation is a relatively slow running ASIC or FPGA. The prototype uses a 16Kb SRAM mat, a capacity on the low end of the optimal energy-delay range, but still achieved area and power overheads of 32% and 23% of the totals respectively. Our projections based on the experimental results show that the reconfiguration overheads can be reduced to below 15% of the area and below 10% of the power by using SRAMs larger than in our prototype design, but still in the near optimum energy-delay block size range. These overheads may reduce even further if we merge the peripheral logic with other necessary logic already present in the SRAM, such as a BIST controller.

Thus, we can build a fast, efficient, reconfigurable memory block by adding only a small amount meta-data and peripheral logic to a basic memory array. The meta-data was a common memory paradigm used across many memory structures. While the peripheral logic blocks were more tailored to individual memory structures, the necessary logic was limited and thus the overhead for hardwired peripheral logic units was small. Where the memory structures did vary significantly is in the way that the mats were connected to each other and to the computation. This led us to use very rich interconnect structures for the inter-mat control and the processor interconnect networks. Similar to conventional memories, the performance and power of reconfigurable memory designs may begin to be dominated not by the memory cores themselves, but rather by the interconnection networks. To that end, we believe that the design of interconnection networks for reconfigurable memories warrants additional scrutiny. Under further study, a simple, low-overhead interconnection topology may emerge that can emulate the necessary interconnect for many memory structures, just as our memory mat does for the memory cores.

Appendix A

SRAM Survey

In Figure 2.4, we plotted the block sizes of a number of contemporary SRAM against the total SRAM capacity. These SRAMs ranged in total capacity from 72Mb to 15Kb, in technologies from 0.65μm to 90nm. However, the majority of the designs used block sizes that only ranged from 16Kb to 128Kb. This supports the conclusions of both Amrutur [24] and Evans [25] that the optimal energy-delay partition size falls within this range. For our reconfigurable memory, we set the mat memory capacity based on the optimal energy-delay block range. Table A.1 below shows the raw data used to generate Figure 2.4. The SRAMs are sorted by total capacity, in descending order.

Table A.1: SRAM survey data

Citation	Capacity (kb)	Block size (kb)	Technology (μm)	Process type
Cho[145]	73728	16	0.10	CMOS
Weiss[146]	24576	24	0.18	CMOS
Zhao[147]	18432	64	0.18	CMOS
Pilo[148]	18432	72	0.18	CMOS
Osada[149]	16384	512	0.13	CMOS
Ishibashi[150]	4096	64	0.25	CMOS
Braceras[151]	4096	36	0.30	CMOS
Ishibashi[152]	4096	64	0.25	CMOS,4T
Bateman[153]	4096	9	0.35	CMOS
Kimura[154]	2048	64	0.65	BiCMOS
Kushiyama[155]	1024	4	0.35	CMOS
Shibata[156]	1024	128	0.30	CMOS/SOI
Shibata[157]	1024	32	0.50	CMOS
Shimizu[158]	288	288	0.18	CMOS
Pelella[159]	288	36	0.50	CMOS
Sato[160]	256	32	0.40	BiCMOS
Akiyoshi[161]	144	36	0.09	CMOS
Mori[39]	32	1	0.25	CMOS
Lu[162]	15	4	0.50	CMOS

Bibliography

[1] P. Silverman, “Who Can Afford Advanced Lithography?” *Microlithography World*, Nov. 2003.

[2] R. Merritt, “Gurus Urges IC Design Overhaul,” *EE Times*, Sept. 2, 2003.

[3] N. Zhang and R. Brodersen, “The Cost of Flexibility in Systems on a Chip Design for Signal Processing Applications,” http://bwrc.eecs.berkeley.edu/Classes/EE225C/Papers/arch_design.doc, 2002.

[4] N. Tredennick and B. Shimamoto, “The death of microprocessors,” *Embedded Systems Programming*, Aug. 11, 2004.

[5] Xilinx Inc., ”Using the Virtex Block SelectRAM+ Features,” http://www.xilinx.com/bvdocs/appnotes/xapp130.pdf, *Application Note XAPP130*, Dec. 18, 2000.

[6] Altera Inc., ”TriMatrix Embedded Memory Blocks in Stratix II Devices,” *Stratix II Device Handbook*, vol. 2, chapt. 2, http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf, Document Number SII5V2-1.3, July 2004.

[7] S. Hauck, *et al.*, “The Chimera Reconfigurable Functional Unit”, *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[8] J. Hauser and J. Wawrzynek, “Garp: a MIPS Processor With a Reconfigurable Coprocessor,” *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[9] http://www.stretchinc.com

[10] K. Sankaralingam, *et al.*, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," *Proceedings of the International Symposium on Computer Architecture*, 2003.

[11] R. Krashinsky, *et al.*, "The vector-thread architecture," *International Symposium on Computer Architecture*, June 2004.

[12] M. Taylor, *et al.*, "The RAW microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, Mar/Apr 2002.

[13] K. Mai *et al.*, "Smart Memories: A Modular Reconfigurable Architecture," *Proceedings, International Symposium on Computer Architecture*, pp. 161-71, June 2000.

[14] Pact Corp.,"The XPP white paper," http://www.pactcorp.com/xneu/download/xpp_white_paper.pdf, Mar. 27, 2002.

[15] D. Parlour, "The Reality and Promise of Reconfigurable Computing in Digital Signal Processing," *Tutorial, IEEE International Solid-State Circuits Conference*, Feb. 2004.

[16] K. Wu and Y. Tsai, "Structured ASIC, evolution or revolution?," *International Symposium on Physical Design*, April 2004.

[17] R. Sites. "It's the Memory, Stupid!" *Microprocessor Report*, pages 19-20, August 5, 1996.

[18] C. Zhang, *et al.*, "A highly configurable cache architecture for embedded systems," *Proceedings, International Symposium on Computer Architecture*, June 2003.

[19] D. Patterson, *et al.*, "Intelligent RAM (IRAM): Chips that remember and compute," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1997.

[20] A. Khan, *et al.*, "A 150 MHz graphics rendering processor with 256Mb embedded DRAM," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 2001.

[21] B. Dipert, "Cutting-edge consoles target the television," *EDN*, December 12, 2001.

[22] W. Leung, *et al.*, "The ideal SoC memory: 1T-SRAM," *Proceedings of the 13th Annual IEEE International ASIC/SOC Conference*, Sept. 2000.

[23] D. Fried, *et al.*, "Aggressively scaled ($0.143\mu m^2$) 6T-SRAM cell for the 32 nm node and beyond," *Technical Digest Electron Devices Meeting*, Dec. 2004.

[24] B. Amrutur and M. Horowitz, "Speed and Power Scaling of SRAM's," *IEEE Journal of Solid-State Circuits*, Feb. 2000.

[25] R. Evans, "Energy consumption modeling and optimization for SRAMs," *Ph.D. dissertation*, Dept. of Electrical and Computer Engineering, North Carolina State University, July 1993.

[26] T. Wada, *et al.*, "An analytical access time model for on-chip cache memories," *IEEE Journal of Solid-State Circuits*, Aug. 1992.

[27] M. Yoshimoto, *et al.*, "A 64kb CMOS RAM with divided word line structure," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1983.

[28] T. Hirose, *et al.*, "A 20ns 4Mb CMOS SRAM with hierarchical word decoding architecture," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1990.

[29] K. Osada, *et al.*, "A 2ns access, 285MHz, two-port cache macro using double global bit-line pairs," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1997.

[30] R. Evans and P. Franzon, "Energy consumption modeling and optimization for SRAMs," *IEEE Journal of Solid-State Circuits*, May 1995.

[31] B. Amrutur, "Design and analysis of fast low power SRAMs," *Ph.D. dissertation*, Dept. of Electrical Engineering, Stanford University, Aug. 1999.

[32] I. Sutherland *et al.*, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann, January 1999.

[33] T. Chappell, *et al.*, "A 2-ns cycle, 3.8-ns access 512kb CMOS ECL SRAM with a fully pipelined architecture," *IEEE Journal of Solid-State Circuits*, Nov. 1991.

[34] B. Amrutur and M. Horowitz, "Fast low-power decoders for RAMs," *IEEE Journal of Solid-State Circuits*, Oct. 2001.

[35] B. Amrutur and M. Horowitz, "A replica technique for wordline and sense control in low-power SRAMs," *IEEE Journal of Solid-State Circuits*, Aug. 1998.

[36] H. Nambu, *et al.*, "A 1.8ns access, 550MHz 4.5Mb CMOS SRAM," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1998.

[37] K. Sakai, *et al.*, "A 15ns 1-Mbit CMOS SRAM," *IEEE Journal of Solid-State Circuits*, Oct. 1988.

[38] M. Matsumiya, *et al.*, "A 15ns 16-Mb CMOS SRAM with interdigitated bit-line architecture," *IEEE Journal of Solid-State Circuits*, Nov. 1992.

[39] T. Mori, it et al., " A 1V 0.9mW at 100MHz 2k*16b SRAM utilizing a half-swing pulsed decoder and write-bus architecture in 0.25μm dual-Vt CMOS" *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1998.

[40] K. Mai, *et al.*, "Low-Power SRAM Design Using Half-Swing Pulse-Mode Techniques," *IEEE Journal of Solid-State Circuits*, Nov. 1998.

[41] J. Alowersson, *et al.*, "SRAM cells for low-power write in buffer memories," *IEEE Symposium on Low Power Electronics*, Oct. 1995.

[42] H. Mizuno, *et al.*, "Driving source-line cell architecture for sub-1-V high-speed low-power applications," *IEEE Journal of Solid-State Circuits*, April 1996.

[43] M. Margala, *et al.*, "Low-power SRAM circuit design," *Records of the 1999 IEEE International Workshop on Memory Technology, Design and Testing*, Aug. 1999.

[44] J. Wang, *et al.*, "Low-power embedded SRAM with the current-mode write technique," *IEEE Journal of Solid-State Circuits*, Jan. 2000.

[45] K. Kanda, *et al.*, "90% write power-saving SRAM using sense-amplifying memory cell," *IEEE Journal of Solid-State Circuits*, June 2004.

[46] S. Heo, *et al.*, "Dynamic Fine-Grain Leakage Reduction using Leakage-Biased Bitlines," *International Symposium on Computer Architecture*, May 2002.

[47] K. Osada, *et al.*, "Universal-Vdd 0.65-2.0-V 32-kB cache using a voltage-adapted timing-generation scheme and a lithographically symmetrical cell," *IEEE Journal of Solid-State Circuits*, Nov. 2001

[48] K. Agawa, *et al.*, "A bitline leakage compensation scheme for low-voltage SRAMs," *IEEE Journal of Solid-State Circuits*, May 2001.

[49] Y. Ye, *et al.*, "A 6-GHz 16-kB L1 cache in a 100-nm dual-Vt technology using a bitline leakage reduction (BLR) technique," *IEEE Journal of Solid-State Circuits*, May 2003.

[50] A. Alvandpour, *et al.*, "Bitline leakage equalization for sub-100nm caches," *Proceedings of the 29th European Solid-State Circuits Conference*, Sept. 2003.

[51] S. Tachibana, *et al.*, "A 2.6-ns wave-pipelined CMOS SRAM with dual-sensing-latch circuits," *IEEE Journal of Solid-State Circuits,* April 1995.

[52] S. Schuster, *et al*, "A 15-ns CMOS 64K RAM," *IEEE Journal of Solid-State Circuits,* Oct. 1986.

[53] T. Higuchi, *et al.*, "A 500MHz synchronous pipelined 1Mbit CMOS SRAM," *Technical Report of IEICE*, May 1996, pp. 9-14, (*in Japanese*).

[54] R. Ho, *et al.*, "Efficient On-Chip Global Interconnects," *Digest of Technical Papers, Symposium on VLSI Circuits*, June 2003.

[55] H. Zhang *et al.*, "Low-swing on-chip signaling techniques," *IEEE Transactions on VLSI*, pp. 414-419, April 1993.

[56] R. Ho, "On-chip wires: scaling and efficiency," *Ph.D. dissertation*, Dept. of Electrical Engineering, Stanford University, Aug. 2003.

[57] B. Nikolic, *et al.*, "Sense Amplifier Based Flip-Flop," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1999.

[58] L. Benini, *et al.*, "Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques," *ACM Transactions on Embedded Computing Systems*, Feb. 2003.

[59] F. Balasa, *et al.*, "Background memory area estimation for multidimensional signal processing systems," *IEEE Transactions on VLSI Systems*, June 1995.

[60] J. Stinson and S. Rusu, "A 1.5GHz third generation Itanium processor," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 2003.

[61] R. Banakar, *et al.*, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems," *International Symposium on Hardware/Software Codesign*, May 2002..

[62] R. Lee and M. Smith, "Media processing: a new design target," *IEEE Micro*, Aug. 1996.

[63] D. Zucker, *et al.*, "Improving performance for software MPEG players," *Proceedings of Compcon*, 1996.

[64] K. Diefendorff and P. Dubey, "How multimedia workloads will change processor design," *IEEE Computer*, Sept. 1997.

[65] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the International Symposium on Computer Architecture*, May 1990.

[66] S. Rixner, *et al.*, "A Bandwidth Efficient Architecture for Media Processing," *International Symposium on Microarchitecture*, 1998.

[67] S. Palacharla, *et al.*, "Evaluating stream buffers as a secondary cache replacement," *In Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24-33, 1994.

[68] K. Farkas, *et al.*, "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors?" *In Proceedings of the First International Conference on High Performance Computer Architecture*, pages 78-89, Jan. 1995.

[69] W. Dally, *et al.*, "Merrimac: supercomputing with streams," *Proceedings of the ACM/IEEE Supercomputing Conference SC2003*, Nov. 2003.

[70] N. Jayasena, *et al.*, "Stream register files with indexed access," *Proceedings of the Tenth International Symposium on High Performance Computer Architecture*, Feb. 2004.

[71] E. Kilgariff and R. Fernando, "The Geforce 6 series architecture," *GPU Gems 2*, Addison-Wesley, 2005.

[72] D. Albonesi, "Selective cache ways: on-demand cache resource allocation," *International Symposium on Microarchitecture*, 1999.

[73] D. Albonesi, *et al.*, "Dynamically Tuning Processor Resources with Adaptive Processing," *IEEE Computer*, Dec. 2003.

[74] R. Balasubramonian, *et al.*, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," *International Symposium on Microarchitecture*, Dec. 2000.

[75] P. Panda, *et al.*, "Data memory organization and optimizations in application-specific systems," *IEEE Design and Test of Computers*, May-June 2001.

[76] P. Panda, *et al.*, "Local memory exploration optimization in embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Jan. 1999.

[77] A. Veidenbaum, *et al.*, "Adapting cache line size to application behavior," *International Conference on Supercomputing*, 1999.

[78] P. Ranganathan, *et al.*, "Reconfigurable caches and their application to media processing," *Proceedings, International Symposium on Computer Architecture*, 2000.

[79] M. Qureshi, *et al.*, "The V-way cache: demand-based associativity via global replacement," *International Symposium on Computer Architecture*, June 2005.

[80] S. Kaxiras, *et al.*, "Cache decay: exploiting generational behavior to reduce cache leakage power," *International Symposium on Computer Architecture*, June 2001.

[81] N. Kim, *et al.*, "Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction," *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture*, Nov. 2002.

[82] A. Malik, *et al.*, "A low power unified cache architecture providing power and performance flexibility," *International Symposium on Low Power Electronics and Design*, 2000.

[83] E. Witchel, *et al.*, "Direct addressed caches for reduced power consumption," *34th International Symposium on Microarchitecture*, Dec. 2001.

[84] Texas Instruments Inc., "TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide (Rev. B)," http://www-s.ti.com/sc/psheets/spru609b/spru609b.pdf, TI Literature Number: SPRU609B, June 2004.

[85] B. Ackland, *et al.*, "A single chip, 1.6-billion, 16-b MAC/s multiprocessor DSP," *IEEE Journal of Solid-State Circuits*, March 2000.

[86] H. Kim, *et al.*, "A Reconfigurable Multifunction Computing Cache Architecture," *IEEE Transactions on Very Large Scale Integration*, Aug. 2001.

[87] V. Srini, *et al.*, "Reconfigurable memory module in the RAMP system for stream processing," *Proceedings of International Symposium on Computer Architecture Workshop*, June 2001.

[88] T. Ngai, *et al.*, "An SRAM-programmable field-configurable memory," *IEEE Custom Integrated Circuits Conference*, 1995.

[89] F. Heile, *et al.*, "Programmable memory blocks supporting content-addressable memory," *International Symposium on Field Programmable Gate Arrays*, 2000.

[90] S. Guccione, *et al.*, "A reconfigurable content addressable memory," *Proceedings of the 15th International Parallel and Distributed Processing Workshops (IPDPS)*, May 2000.

[91] Personal communications with Manoj Chirania, Xilinix Inc., Jan. 2004.

[92] S. Scott, "Synchronization and communication in the T3E multiprocessor," *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[93] L. Hammond, "Hydra: a chip multiprocessor with support for speculative thread-level parallelization," *Ph.D. dissertation*, Dept. of Electrical Engineering, Stanford University, Mar. 2002.

[94] M. Horowitz and W. Dally, "How Scaling Will Change Processor Architecture" *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 2004.

[95] D. Sager, *et al.*, "A 0.18μm CMOS IA32 Microprocessor with a 4GHz Integer Execution Unit," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 2001.

[96] M. Matson, *et al.*, "Circuit Implementation of a 600MHz Superscalar RISC Microprocessor," *Proceedings of the International Conference on Computer Design*, October, 1998

[97] S. Hsu, *et al.*, "A 4.5-GHz 130-nm 32-KB L0 cache with a leakage-tolerant self reverse-bias bitline scheme," *IEEE Journal of Solid-State Circuits*, May 2003.

[98] D. Pham, *et al.*, "Design and implementation of a first-generation CELL processor," *International Solid-State Circuits Conference*, Feb. 2005.

[99] S. Dhong, *et al.*, "A 4.8GHz fully pipelined embedded SRAM in the streaming processor of a CELL processor," *International Solid-State Circuits Conference*, Feb. 2005.

[100] S. Rixner, *et al.*, "Register organization for media processing," *International Symposium on High Performance Computer Architecture*, Jan. 2000.

[101] O. Ergin, *et al.*,"A Circuit-Level Implementation of Fast, Energy-Efficient CMOS Comparators for High-Performance Microprocessors," *IEEE International Conference on Computer Design*, Sept. 2002.

[102] C. Wang, *et al.*, "High fan-in dynamic CMOS comparators with low transistor count," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Sept. 2003

[103] C. Huang, *et al.*, "High-performance and power-efficient CMOS comparators," *IEEE Journal of Solid-State Circuits*, Feb. 2003

[104] H. Mizuno, *et al.*, "A 1-V, 100-MHz, 10-mW cache using a separated bit-line memory hierarchy architecture and domino tag comparators," *IEEE Journal of Solid-State Circuits*, Nov. 1996

[105] J. Hennessy and D. Patterson. *Computer Architecture, a Quantitative Approach*. 2nd Ed. 1996.

[106] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann,2004.

[107] J. Duato, *et al.*, *Interconnection Networks*, Morgan Kaufmann, 2003.

[108] C. Seitz, "Let's route packets instead of wires," *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, 1990.

[109] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," *Proceeding of the ACM/IEEE Design Automation Conference*, June 2001.

[110] L. Benini and G. DeMicheli, "Networks on chips: a new SoC paradigm," *IEEE Computer*, Jan. 2002.

[111] P. Magarshack and P. Paulin, "System-on-chip beyond the nanometer wall," *Proceedings of the ACM/IEEE Design Automation Conference*, June 2003.

[112] S. Dutta, *et al.*, "High-performance crossbar interconnect for a VLIW video signal processor," *Ninth Annual IEEE International ASIC Conference and Exhibit, Proceedings*, Sept. 1996.

[113] A. Boxer, "Where buses cannot go," *IEEE Spectrum*, Feb. 1995.

[114] Y. Zhang, *et al.*, "An alternative architecture for on-chip global interconnect: segmented bus power modeling," *Conference Record of the Thirty-Second Asilomar Conference on Signals, Systems and Computers*, Nov. 1998.

[115] J. Chen, *et al.*, "Segmented bus design for low-power systems," *IEEE Transactions on Very Large Scale Integration Systems*, March 1999.

[116] J. Plosila,*et al.*, "Implementation of a self-timed segmented bus," *IEEE Design and Test of Computers*, Nov.-Dec. 2003.

[117] V. Lahtinen, *et al.*, "Comparison of synthesized bus and crossbar interconnection architectures," *Proceedings of the International Symposium on Circuits and Systems*, May 2003.

[118] R. Naik, *et al.* "Large integrated crossbar switch," *Proceedings of the Seventh Annual IEEE International Conference on Wafer Scale Integration*, Jan. 1995.

[119] K. Choi and W. Adams, "VLSI implementation of a 256256 crossbar interconnection network," *Proceedings of the Sixth International Parallel Processing Symposium*, March 1992.

[120] N. McKeown, *et al.* "Tiny Tera: a packet switch core," *IEEE Micro*, Jan.-Feb. 1997.

[121] A. Lines, "Asynchronous interconnect for synchronous SoC design," *IEEE Micro* Jan-Feb. 2004.

[122] M. Borgatti, *et al.*, "A multi-context 6.4Gb/s/channel on-chip communication network using 0.18μm Flash-EEPROM switches and elastic interconnects," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 2003.

[123] S. Lee, *et al.*, "An 800MHz star-connected on-chip network for application to systems on a chip," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 2003.

[124] K. Lee, *et al.*, "A 51mW 1.6GHz on-chip network for low-power heterogeneous SoC platform," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 2004.

[125] J. Labrousse and G. Slavenburg, "A 50 MHz microprocessor with a very long instruction word architecture," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1990.

[126] L. Lin, *et al.*, "CCC: crossbar connected caches for reducing energy consumption of on-chip multiprocessors," *Proceedings of the Euromicro Symposium on Digital System Design*, Sept. 2003.

[127] K. Sankaralingam, *et al.*, "Routed inter-ALU networks for ILP scalability and performance," *Proceedings of the 21st International Conference on Computer Design*, Oct. 2003.

[128] E. Fetzer, *et al.*, "A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor," *IEEE Journal of Solid-State Circuits*, Nov. 2002.

[129] P. Gupta and N. McKeown, "Designing and implementing a fast crossbar scheduler," *IEEE Micro*, Jan.-Feb. 1999.

[130] J. Liang, *et al.* "ASOC: a scalable, single-chip communications architecture," *Proceeding of the International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2000.

[131] K. Lee, *et al.*, "A high-speed and lightweight on-chip crossbar switch scheduler for on-chip interconnection networks," *Conference on European Solid-State Circuits*, Sept. 2003.

[132] K. Lee, *et al.*, "A distributed crossbar switch scheduler for on-chip networks," *Proceedings of the IEEE Custom Integrated Circuits Conference*, Sept. 2003.

[133] M. Sinha and W. Burleson, "Current-sensing for crossbars," *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*, Sept. 2001.

[134] P. Wijetunga, "High-performance crossbar design for system-on-chip," *Proceedings of the IEEE International Workshop on System-on-Chip for Real-Time Applications*, 2003.

[135] H. Wang, *et al.*, "Power-driven design of router microarchitectures in on-chip networks," *Proceedings of the International Symposium on Microarchitecture*, 2003.

[136] A. Stratakos, *et al.*, "A low voltage CMOS DC-DC converter for a portable battery-operated system,"*Proceedings of IEEE Power Electronics Specialists Conference*, June 1994.

[137] Y. Zhang and M. Irwin, "Power and performance comparisons of crossbars and buses as on-chip interconnect structures," *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers*, Oct. 1999.

[138] E. Geethanjali, *et al.*, "An analytical power estimation model for crossbar interconnects," *15th Annual IEEE International ASIC/SOC Conference*, Sept. 2002.

[139] R. Heald, *et al.*, "64 KByte sum-addressed-memory cache with 1.6 ns cycle and 2.6 ns latency," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 1998.

[140] K. Mai, *et al.*, "Architecture and Circuit Techniques for a Reconfigurable Memory Block," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, Feb. 2004.

[141] K. Mai, *et al.*, "Architecture and circuit techniques for a 1.1GHz 16kb reconfigurable memory in 0.18μm CMOS," *IEEE Journal of Solid-State Circuits*, Jan. 2005.

[142] S. Vangal, *et al.*, "5-GHz 32-bit integer execution core in 130-nm dual-Vt CMOS," *IEEE Journal of Solid-State Circuits*, Nov. 2002.

[143] R. Ho, *et al.*, "Applications on On-Chip Samplers for Test and Measurement of Integrated Circuits," *Digest of Technical Papers, Symposium on VLSI Circuits*, pp. 138-9, June 1998.

[144] R. Ho, *et al.*, "The Future of Wires," *Proceedings of the IEEE*, April 2001.

[145] U. Cho, *et al.*, "A 1.2 V 1.5 Gb/s 72 Mb DDR3 SRAM," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 2003.

[146] D. Weiss, *et al.*, "An on-chip 3MB subarray-based 3rd level cache on an itanium microprocessor," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 2002.

[147] C. Zhao, *et al.*, "An 18-Mb, 12.3-GB/s CMOS pipeline-burst cache SRAM with 1.54 Gb/s/pin," *IEEE Journal of Solid-State Circuits*, Nov. 1999.

[148] H. Pilo, *et al.*, "An 833 MHz 1.5 W 18 Mb CMOS SRAM with 1.67 Gb/s/pin," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 2000.

[149] K. Osada, *et al.*, "16.7 fA/cell tunnel-leakage-suppressed 16 Mb SRAM for handling cosmic-ray-induced multi-errors," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 2003.

[150] K. Ishibashi, *et al.*, "A 300 MHz 4-Mb wave-pipeline CMOS SRAM using a multi-phase PLL," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 1995.

[151] G. Braceras, *et al.*, "A 350 MHz 3.3 V 4 Mb SRAM fabricated in a 0.3 μm CMOS process," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 1997.

[152] K. Ishibashi, *et al.*, "A 6-ns 4-Mb CMOS SRAM with offset-voltage-insensitive current sense amplifiers," *IEEE Journal of Solid-State Circuits*, April 1995.

[153] B. Bateman, *et al.*, "A 450 MHz 512 kB second-level cache with a 3.6 GB/s data bandwidth," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 1998.

[154] T. Kimura, *et al.*, "Design of 1.28-GB/s high bandwidth 2-Mb SRAM for integrated memory array processor applications," *IEEE Journal of Solid-State Circuits*, June 1995.

[155] N. Kushiyama, *et al.*, "A 295 MHz CMOS 1 M (256) embedded SRAM using bidirectional read/write shared sense amps and self-timed pulsed word-line drivers," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 1995.

[156] N. Shibata, *et al.*, "A 2-V 300-MHz 1-Mb current-sensed double-density SRAM for low-power 0.3-μm CMOS/SIMOX ASICs," *IEEE Journal of Solid-State Circuits*, Oct. 2001.

[157] N. Shibata, *et al.*, "A 1-V, 10-MHz, 3.5-mW, 1-Mb MTCMOS SRAM: with charge-recycling input/output buffers," *IEEE Journal of Solid-State Circuits*, June 1999.

[158] H. Shimizu, *et al.*, "A 1.4 ns access 700 MHz 288 kb SRAM macro with expandable architecture," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 1999.

[159] M. Pelella, *et al.*, "A 2 ns access, 500 MHz 288 Kb SRAM macro," *Digest of Technical Papers, Symposium on VLSI Circuits*, June 1996.

[160] H. Sato, *et al.*, "A 3.6 mW 1.4 V SRAM with non-boosted, vertical bipolar bitline contact memory cell," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 1998.

[161] H. Akiyoshi, *et al.*, "A 320ps access, 3GHz cycle, 144Kb SRAM macro in 90nm CMOS technology using an all-stage reset control signal generator," *Digest of Technical Papers, International Solid-State Circuits Conference*, Feb. 2003.

[162] P. Lu, *et al.*, "A 15Kb 1.5 ns Access On-chip Tag SRAM," *Proceedings of Technical Papers, International Symposium on VLSI Technology, Systems, and Applications*, June 1997.