Design of the Mosaic Processor

Christopher Lutz

Computer Science Department
California Institute of Technology

5129:TR:84

Design of the Mosaic Processor

by

Christopher Lutz

In Partial Fulfillment of the Requirements for the
Degree of Master of Science

May 1984

5129:TR:84
Computer Science
California Institute of Technology
Pasadena, CA 91125

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

Contents

1.	Introduction	1
2.	Top-level View	1
3.	Chronology	2
4.	Processor Organization	3
5.	Datapath	6
6.	Ports	8
7.	Controller	9
8.	Instruction Set	11
9.	Microcode	12
10.	From Version A to Version B	13
11.	Sample Instruction Execution	14
12.	Memory	17
13.	Circuit Design	18
14.	Design Tools	19
15.	Testing	20
16.	Acknowledgements	21
17.	References	22
	APPENDIX A: Processor Version A	
	Instruction Set	23
	Microcode Source	26
	Circuit Diagrams	29
	APPENDIX B: Processor Version B	
	Instruction Set	44
	Microcode Source	48
	Circuit Diagrams	55
	APPENDIX C: Selected Layout	59

Design of the Mosaic Processor[1]

1. Introduction

The Mosaic element is a fast single chip nMOS computer designed to be used in groups for concurrent computation experiments. Each element contains a 16-bit processor, 4 input ports, 4 output ports, and read-write memory. This thesis describes the design of the processor and ports in detail. The memory section, mentioned here only briefly, has been designed separately and will later be incorporated on the same chip with the processor and ports.

Myriads of Mosaic elements can be connected together by their ports in a variety communication plans, such as a tree, mesh, shuffle, chordal ring, or cube connected cycle, to form a family of specialized, high performance, concurrent, and programmable computing engines. Mosaic is one of several system building experiments in concurrent computation underway at Caltech; a discussion of the rationale, programming style, and applications of these machines is offered in [Seitz84]. In addition to its end use as a component for experiments with concurrent computing engines, Mosaic has been an interesting vehicle for numerous adventures in VLSI design, design tools, and testing.

The principle objectives in designing the processor have been speed, simplicity, and the flexibility to serve a wide variety of applications, anticipated and not.

2. Top-level view

It appears that most of the silicon area in multiple-instruction multiple-data (MIMD) ensemble machines should be devoted to memory for the best tradeoff between performance and generality. In cosmic cube, a larger grain size machine of similar style to Mosaic, the fraction of the element complexity devoted to memory is about 75%. With the precondition that a complete Mosaic element fit on a single chip, and using today's MOSIS nMOS fabrication with 1.5 micron lambda (3 micron feature size) on chips 6 mm square, the complexity of today's Mosaic element is limited to 4000 by 4000 lambda, or 16 million square lambda (MSL). This area is apportioned with about 2.5 MSL for the processor and ports, 1 MSL for the pad frame, and 12 MSL (75%) for memory and its interconnect.

[1][Lutz,Rabin,Seitz&Speck83] is a short precursor to this thesis.

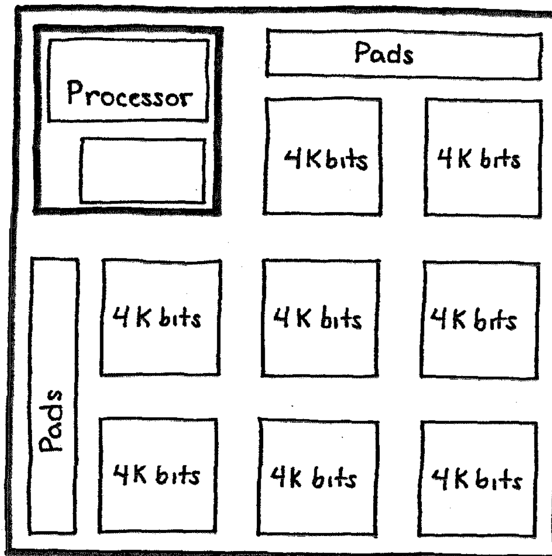


Figure 1: Mosaic Element Floorplan

The floorplan of the Mosaic element places the processor and ports in one corner, and fills the rest of whatever chip area is available with copies of 4096-bit memory modules arranged so that the address and data buses can be run in between. For a 16 MSL element, the chip has the floorplan shown in figure 1.

3. Chronology

The original models for this project, from which everything else evolved, were (1) Sally Browning's research on algorithms for a programmable tree machine [Browning80a, Browning80b, Browning&Seitz81], and (2) the "OM" described in [Mead&Conway80]. Mosaic started out as a tree machine element, but we have since come to see it as a building block for a variety of fine grain ensemble machines with connection plans up to degree 4. The influence of the OM layout style on the floorplan and many details of the ALU of the Mosaic processor will be apparent.

An early attempt to lay out a less ambitious processor with a 4-bit path to off-chip memory used the prototype versions of Earl, a constraint solving geometry and composition tool [Kingsley82]. The early processor and early Earl served as mutual guinea pigs as they were growing up together.

A major redesign followed the decision to incorporate Mosaic's memory on-chip with the processor, rather than use commercially produced RAM

chips. Although specialized semiconductor processes provide higher storage density than processes suitable for the Mosaic processor, putting the storage on-chip with the processor offers the advantages of reduced pin-count, volume, signal energy, driver delay, and cost.

A new processor, featuring a 16-bit data path intended to be connected to on-chip memory, was designed in the 1981-82 academic year, laid out and verified in the summer and fall of 1982, and sent to MOSIS for fabrication at a 4 micron feature size in January 1983. It functioned nearly correctly and at 140 nsec cycle time on first silicon in February 1983. Appendix A contains the instruction set, microcode source, and circuit diagrams for this design, called version A. The processor was subsequently redesigned with additional functions and a faster control PLA; Appendix B describes this latest design, called version B.

In the meanwhile, the on-chip memory sections have been designed, and processor chips have been assembled with fast off-chip memory to make some small prototype Mosaic elements. These elements will be used for programming experiments and software development in anticipation of larger systems to be built with the fully integrated Mosaic elements.

4. Processor organization

Figure 2 is the floorplan of the core of the processor, without the surrounding memory and pad frame. The processor's organization is summarized by the detailed block diagram of figure 3. The processor has two principle components: a datapath/port block, and a controller, each of which is a dense, regular block of layout. The datapath/port block is functionally centered around the processor's single 16-bit internal data bus; it is controlled by signals issued by the PLA-based controller. The following sections describe these components (for version A) in detail. Then section 10 describes the changes that yielded version B.

The instruction register (I) holds the current macroinstruction and can be latched from the memory data bus on command from the controller. Parts of the instruction register are distributed in several places in the processor depending on the bits needed locally: in the controller, near the flags, and near the register and port selectors. Some of the bits are duplicated in different places.

The Mosaic element is synchronous, with 2 sets of 2-phase non-overlapping clocks supplied externally (figure 4). The clocks are nominally 7 volts (with Vdd = 5 volts) for reasons discussed in section 13. The primary clocks, called φ_1 and φ_2, have minimum high times of roughly 60 nsec and 30 nsec, respectively. The secondary clocks, φ_{1L} and φ_{2L}, are used

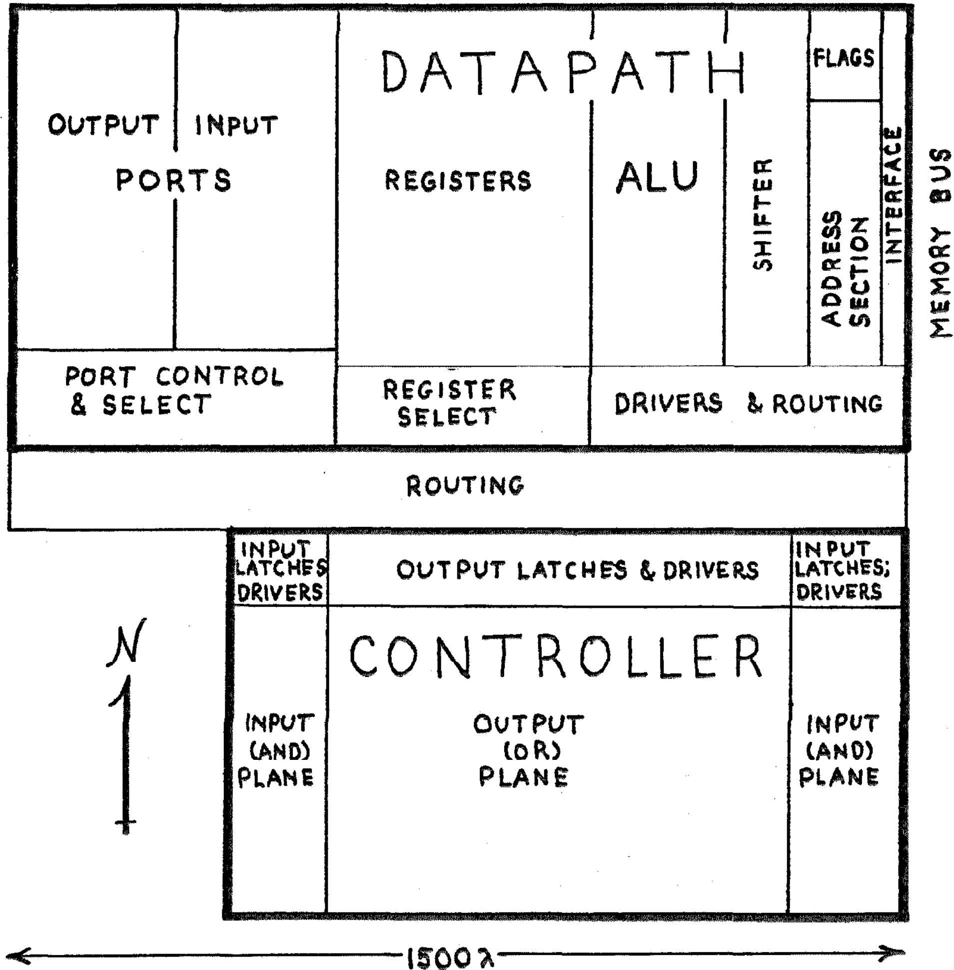


Figure 2: Processor floorplan

principally in the memory sections; version A processors do not use them at all. The memory cycle, processor microcycle, datapath operations, and serial communication cycle occur in parallel in one clock cycle.

The processor design is the result of dozens of iterations through the design of the instruction set, microcode, floorplan, logic design, and circuit design. Thus the rationales for many of the design details are buried in a long history of shuffling and trial-and-error. While many of the design decisions are individually somewhat arbitrary, their justification is that they work well together.

For example, the choice of a single internal data bus sometimes limits the performance, but a slower and more complex controller and address

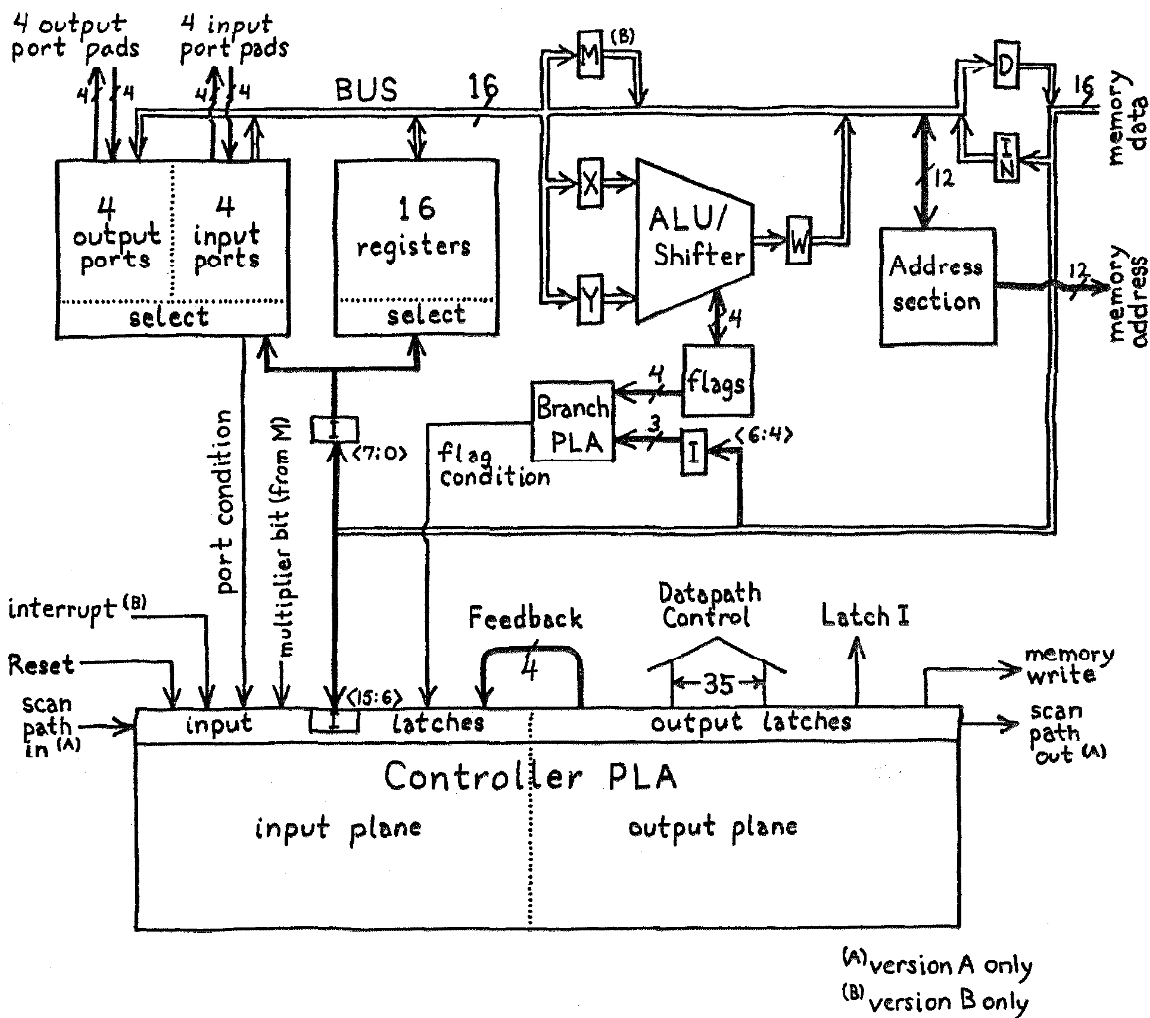


Figure 3: Processor Block Diagram

section would be required to take much advantage of more busses. The processor opts for a more leisurely approach in which simple instructions take 3 cycles: enough to do instruction decode, operand fetch, and operand use on separate cycles. This keeps the per-cycle capabilities of the bus, ALU, controller, ports, address section, and memory are well matched.

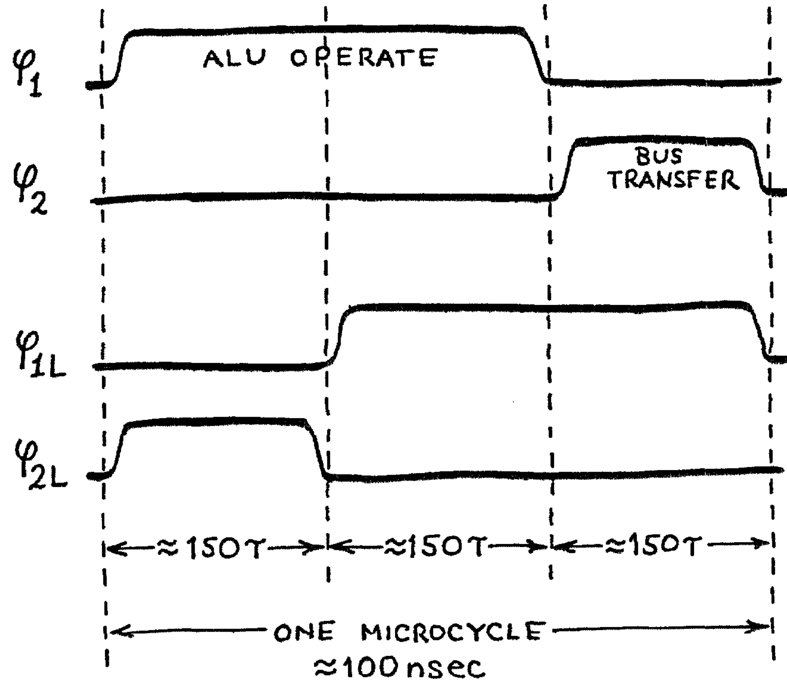


Figure 4: Externally Supplied Clocks

5. Datapath

The datapath contains those parts of the processor that communicate over the internal data bus. The bus is 16 bits wide and runs the length of the datapath. The functional blocks in the datapath are organized in a bit slice pattern, one bit of the bus running through each bit slice, with a bit slice pitch of 34 lambda. During φ_1 of each cycle the bus is precharged and the ALU/shifter computes a new result. φ_2 is used for the bus transfer and the ALU carry chain precharge.

Except in the register section, the control signals, power, and clocks run (vertically) on metal perpendicular to the bus. The register section was more compactly laid out with vertical poly control signals and clocks, bus, and power run horizontally on metal.

The datapath includes sixteen 16-bit registers which are used as general purpose registers in the macroinstruction set. In every cycle, one of the registers may be a bus source or bus destination. The register used is addressed by either the J field (bits 0 through 3) or the K field (bits 4 through 7) of the instruction register, as determined by a signal from the controller. The controller thus cannot specify a particular register directly.

The array is composed of pseudostatic storage cells, refreshed on φ_1.

The ALU/shifter performs in one cycle any of the arithmetic, shift, and logical operations required by the arithmetic instructions. The ALU operands are held in a pair of latches, called X and Y, that are loaded from the bus. The ALU is patterned after the ALU in the OM design [Mead&Conway80], with a pair of function blocks and a precharged pass transistor carry chain. Mosaic uses an exclusive-OR gate at the ALU output rather than the slower and needlessly general result function block used in the OM. Although the ALU does not use carry lookahead, it is optimized to the extent that it is not in the critical timing path.

The ALU result serves as input to a shifter, which uses pass gates to route correctly shifted data to the ALU/shifter output. The shifter can shift or rotate right one bit, rotate right by 4 bits (nibble rotate), or pass the ALU output through unchanged. The shifter could have been placed in parallel with the ALU, rather than in series with it, since the services of both are never required in the same cycle. However, the series organization is preferable because (1) it is simpler, since only one set of latches and flag interface logic is needed, (2) it costs nothing in speed, since computing the overflow flag is the slowest path. This is possible because the pass transistors in the shifter are carefully placed so that the capacitive loads on the most significant (last computed) bits out of the ALU are small when no-shift is performed.

The processor maintains four flags in association with the ALU/shifter. These are the familiar Z (zero result), N (negative result), V (two's complement overflow), and C (carry/not borrow). The C flag is also used as the shift in and/or assigned the shift out in 1-bit shift and rotate instructions. The controller does not sense the values of the flags directly. Instead, a fixed 3-bit field in conditional branch macroinstructions specifies one of eight branch conditions. These three bits, as well as the values of the four flags, are inputs to a small PLA that produces one bit of output, the "flag condition". This bit is an input to the controller, which tests it in performing the conditional branch instructions. Thus the controller is not burdened with computing the flag condition itself. The branch condition codes were assigned carefully so the flag condition PLA requires only six implicants. Since the PLA is so small, it fits neatly next to the flags in the corner of the processor, in a region formed by removing the top four bit slices of address generation. The 4 flags and 12-bit program counter form a 16-bit status word, conveniently located to communicate with the bus.

Every cycle the address section emits a new memory address onto the 12 memory address wires that come out of the right edge of the datapath. The address generation section houses the program counter register (PC), the refresh address register (RA), the current memory address register (A)

and an incrementer. The microcode guarantees that the RA is incremented and issued to the memory at least once every 8 cycles. The processor's performance is not degraded by this refresh task because only memory cycles which would otherwise go to waste are used for refresh cycles.

A 12-bit address is sufficient to address the number of words of memory we can currently place on-chip. If in the future more than 12 bits of address are needed, either the word length of the processor can be increased, or the flags can be moved to a new status word apart from the PC, and the address section lengthened to 16 bits. Neither solution is traumatic.

6. Ports

Mosaic processors communicate with each other through their ports. Each processor has 4 input ports and 4 output ports. Connecting an output port of one processor to an input port of another (not necessarily different) processor forms a two-word FIFO. That is, words can be removed by the processor with the input port in the order in which they are inserted by the processor with the output port, and as many as two more words can be inserted than have been removed. The communication between input and output ports is bit serial at the microcycle rate, about 10MHz.

Mosaic's implementation of the ports requires only a single wire, called the port link, to connect an input to an output port. When a port is not ready to perform a serial transfer, because it is an output port with no data or an input port with unremoved data, it clamps the port link to ground. On the cycle when both ports are ready to perform a transfer, neither processor grounds the port link and an external pullup resistor pulls it to Vdd. Both ports recognize this signal as the "start bit" of a transfer, much as in RS-232 data communications. The next 16 cycles pass the data serially on the port link, and then the ports revert to the clamp-if-not-ready state.

This protocol allows multiple input ports to be connected to the same link: all input ports receive data from the output port beginning on the cycle when all the ports are ready. This feature went unnoticed until after the ports were completely designed.

Each input port is based on a 17-bit serial-in, parallel-out shift register. The input port senses that a transfer is complete and that it should stop shifting when the "start bit" reaches the 17th bit of the shift register. Then the first 16 bits contain the transmitted word. Each output port is based on an 17-bit parallel-in, serial-out shift register. The trailer bit is set to one when a word to be transmitted is loaded from the bus, and is used as a marker to determine when the last bit of the word has been shifted out.

Three bits select a port, always taken from a fixed field of the instruction register (bits 4, 5, and 6). A bit from the ports, the "port condition", indicates whether the selected port is ready to perform a bus operation. If an output port is selected, the controller can direct it to read a new word to transmit from the bus. If an input port is selected, the controller can direct it to drive the bus, or to "advance" (remove a word from the FIFO). The controller is responsible for issuing these signals only when the selected port is ready.

The use of a single fixed port specifier field allows the hardware to be simple, but it made designing a clean instruction set difficult because it requires that all port references, whether for input, output, or testing the condition of a port, coincide in the same field of the instruction word.

Early plans called for a 4-bit message number appended to each transmitted word, which the destination could test with its own input instructions. The message number was later deemed to be insufficient, the design to support it was distressingly complex, and the havoc it caused at the floorplan level was unspeakable. Thus it was discarded, as were several other port designs.

The present is the simplest of all designs considered, but the design as seen from the instruction set has several drawbacks: the number of ports is limited to four; there are no facilities for referencing ports indirectly (instructions must reference them explicitly); and functions such as block transfers, polling for ready ports, and message routing must be done in software at the expense of performance and code space. Furthermore, in this implementation the serial transfer rate is stuck at the processor cycle rate, which requires bounding the transfer distance or slowing the entire processor down to accommodate the slowest communication path in the ensemble.

7. Controller

Each cycle the Mosaic controller computes a new set of signals to control the datapath and ports in the following cycle. The original plans for the controller assume a rather conventional organization in which microcode words are fetched from a ROM, and a new ROM address is computed every cycle by a conglomeration containing an incrementer, multiplexers, and other miscellaneous logic. Most of this complexity disappeared with the realization that a PLA could be efficiently programed to perform most of the controller's function. The controller now required no additional hardware

except input and output latches and an auxiliary PLA for controlling the ALU/shifter. This auxiliary PLA proved to be very troublesome because all the king's horses and all the king's men could not find a placement for it that did not result in large wiring channels and expanses of white space. The auxiliary PLA was finally eliminated by incorporating its function in the main controller PLA. The controller became merely a 2-plane PLA with latches. In most microprocessors the datapath is the most regular part, but in Mosaic the controller is even more regular than the datapath.

The controller is complicated somewhat by a scheme to change its height to width ratio to better fit the space allotted to it: for every controller output, the PLA OR-plane has two outputs and a 2-to-1 multiplexer. The multiplexers are controlled by bit 8 of the instruction register (I<8>), chosen because it allows a large reduction in the number of PLA implicants (outputs from the AND plane). This scheme doubles the number of outputs from the AND plane in return for a roughly 35% reduction in the number of implicants. Programing this "folded" PLA is logically the same as programing a PLA in which implicants are paired, the input conditions of implicants in a pair differing only in bit I<8>.

The AND-plane is split into two parts to make routing of inputs easier. (See the processor floorplan, figure 2.) This splitting requires implicants to be run on metal because poly or diffusion implicant wires would have far too much static voltage drop to allow the processor to work, no matter where the pullups were placed. The PLA outputs are run on diffusion; their pullups are placed opposite the end where the outputs are sensed so that the implicants' static voltage drop does not appear at the sensed end.

The controller also incorporates a shift register through all of its input and output latches. This "scan path" has not been used, although it might have been useful as a diagnostic aid had anything been seriously wrong. It is controlled by auxiliary clocks called φ_a and φ_b which are held stable under normal processor operation.

The controller has 17 inputs: 10 bits from the instruction register, the flag condition, the port condition, the processor reset, and 4 feedback bits (outputs from the controller clocked directly back to the controller input). So little feedback state is needed because much of the state is held in the instruction register (I), and the sequences to implement macroinstructions are short.

Most of the 41 outputs from the controller go to clock-AND drivers (see section 13) that drive control lines into the datapath. Five outputs specify data bus sources; 6 specify bus destinations; 16 control the ALU/shifter and flags; 6 control the address generation section; 4 are feedback terms; and 4 perform miscellaneous functions.

8. Instruction Set

Appendix A specifies the macroinstruction set. All instructions are one word followed optionally by a word of immediate data. In the first instruction word, the two 4-bit fields J and K can each be used to specify one of the general registers. In some instructions, the K field may specify one of the ports or a branch condition instead.

All instructions fetch two operands, X and Y, as specified by the 3-bit MODE field. The X and Y operands in fact correspond to the hardware registers X and Y at the input to the ALU. Two of the MODEs involve ports; their meanings depend on whether field K specifies an input or an output port. Instructions that write to an output port wait until there is room in the FIFO. Instructions that read from an input port wait until there is a word to read, and can optionally "advance" the port (remove the word from the FIFO).

After fetching X and Y all instructions perform the operation specified by the 5-bit OP field. Sixteen compute some function of X and Y and assign the result to a destination as specified by the MODE. The RNR (Rotate Nibble Right) Arithmetic instruction allows access to nibbles (4-bit fields) within words, and performing two successive RNR instructions effects a byte swap. The remaining OPs include a compare, stores, stack manipulations, and flow control. For example, JUMP assigns X to the PC. PUSHJ performs a subroutine call by pushing the PC and flags on a stack, using any register as a stack pointer, and then assigning X to the PC. BRAT and BRAF assign X to the PC if the condition in field K is true (for BRAT) or false (for BRAF). These conditional branches can test the state of the ports individually, the flags individually, or signed and unsigned relations computed from the flags.

The richness of this instruction set is justified largely by the code compactness it offers in its environment of scarce on-chip memory. Perhaps the greatest code space inefficiency is in the lack of short branches: branch instructions take a full word of immediate value to specify the address rather than a short offset, as in a PDP-11. Also, the lack of byte addressing requires the use of full words to store bytes, or emulation of byte addressing.

On reset, the processor begins executing instructions starting from memory location zero, which is assumed to contain ROM for an initialization program.

9. Microcode

The speed, simplicity, and compactness of this design owe much to the realization that the controller need be nothing more than an PLA with latches. A PLA is not merely sufficient; it is convenient and easy to program for an instruction set such as this in which microinstruction sequences are short but heavily branched based on varying fields in the input to the controller.

Each implicant in the PLA is viewed as a word of microcode. More than one word of microcode can be active (that is, more than one implicant can be TRUE) in any given cycle. Usually only one word is active at a time, but there are important exceptions. In these cases, the outputs are partitioned into disjoint sets, such that each word has no TRUE outputs (transistors in the OR plane) outside its set. Thus this controller does not take advantage of ORing of the outputs for the active words, although we might reasonably have done so if it appeared to offer much advantage. The effect of multiple active words used in this restricted manner is like that of multiple disjoint PLAs, but the physical layout retains the regularity of one PLA. In return for this restriction, the absolute true/complemented sense of the individual outputs is irrelevant, the microcode assembler and assembly language is simpler, and the microcode is easier to understand.

An unpleasant feature of writing microcode for a PLA is that words which are active on the logical OR of input conditions (other than the conjunction implicit in don't-care bits) must be implemented with multiple implicants, one for each condition being ORed, and the outputs duplicated for each implicant. Unless care is taken, implicant groups of this sort tend to hog the microcode space. Careful encoding of the macroinstruction set is a partial solution, and the problem is certainly less severe than if the controller had been ROM-based.

A simple microcode assembler, written in SIMULA, reads the source microcode and assembles it into an runtime data structure. From here the assembler can output the code in any of several formats, including Earl source code, and a table of bits for visual checking. The whole process takes 10 CPU seconds on a DEC-20. The assembler also contains an ad hoc register-transfer level (and sometimes gate level) simulator of the processor. This simulator served as an initial debugger for the processor design, and is still the initial proving ground for modifications in the processor and its microcode.

10. From Version A to Version B

The version A processor has been fabricated numerous times at various feature sizes. The version B processor has not yet been completely assembled or fabricated, but is planned for future fabs. This section describes their differences. It illustrates the kind of change that occurred many times in the design history: simplifications or improvements that retain most of the original parts and techniques. This long history of incremental changes is largely responsible for the present design's compactness, speed, and simplicity.

The following changes lead from version A to version B:

A special purpose register, the Multiplier/Product (M), is added to the datapath in association with the ALU. It allows the processor to perform a multiply step in one cycle. The multiplier, initially contained in M, is shifted out and tested by the controller one bit at a time. The least significant bits of the product are at the same time shifted into M. A 16-bit shift register also added to the datapath is used by the controller as an auxiliary finite state machine to count multiply steps. The multiply macroinstruction produces a 32-bit unsigned product in 21 cycles.

The controller circuit design is redone to use precharging and clock-AND drivers, rather than the static design which made the version A controller the speed-limiting component. The controller is logically simplified by eliminating the "folding" technique described in section 7. The controller now has 20 inputs, 49 outputs, and about 120 implicants.

The instruction set is changed to include a set of MOVEs in which 3-bit fields specify any of 8 sources and 8 destinations. The two MODEs which involve the ports are eliminated to make room for the MOVEs. Now all port references must be made with MOVE instructions. The multiply and several additional single-cycle arithmetic instructions are added.

The processor now handles simple external interrupts. When a one-cycle pulse appears on the interrupt pin, the processor completes the instruction in progress, saves the PC and flags in memory location −2, and takes the contents of location −3 as the interrupt service address. These interrupts can be used, for example, to provide an ensemble of processors with periodic interrupts. Periodic interrupts are useful for decoupling communications from procesing, (e.g., to implement automatic message routing, and to buffer large blocks of data) and to give the processor a sense of time (e.g., for heuristic searches). Earlier plans called for an interrupt-generating counter to be placed in each processor. Although the design and layout of the interrupt counter were completed, it was replaced with the simpler and probably more useful external interrupt.

If the interrupt pulse is at least 26 cycles (long enough for multiply to finish and the controller to note that the pulse is persisting longer than a cycle) the processor performs a "soft reset", which completes the instruction in progress, saves the PC and flags in location −1 and then sets the PC to zero, where reset ROM is located. Soft reset differs from hard reset in that soft reset allows the instruction in progress to finish, but cannot force the controller out of illegal states. Soft reset can be used to save the state of a Mosaic ensemble. The ensemble can be restarted later with the same state, except for the exact phase relationships of port transfers and instructions in different processors. This feature can be used as a diagnostic aid, to allow periodic checkpointing of long-running tasks, or to swap tasks and thus allow time-sharing of an ensemble.

In order to guarantee interrupt service in bounded time, the port-wait states must be interruptable. The microcode thus refetches and restarts any port input or output instruction that cannot be completed immediately. Since the instruction register (I) is now guaranteed to be latched periodically, the pseudostatic cells of version A can be replaced with dynamic nodes. The flags are now also writable from the bus, in order to allow return from interrupt.

The major additions yielding version B (multiply, new controller, complex MOVEs, and interrupts) are essentially independent and any subset of them could reasonably be implemented. They are lumped together as "version B" for convenience of presentation.

11. Sample Instruction Execution

In order to illustrate some features of the microcode programming style and processor timing, this section presents a long-winded blow-by-blow description of the execution of a sample (version B) macroinstruction. Figure 5 shows the assembly of the macroinstruction "**ADD #7,R1,R2**", the 4 microcode words required to execute it, and the behavior of various parts of the processor in the vicinity of its execution. This instruction adds immediate data 7 to the contents of register 1, and stores the result in register 2. The instruction executes in 3 cycles, corresponding to the first, second, and last two microcode words (the last two are active simultaneously).

The tokens "**.decode**", "**.get**", and "**.go**" are mnemonics for feedback states; they appear both in the input conditions and in the next state outputs. The first microcode word, "**DECODE:**", is in fact the first word of every instruction. It becomes active any time the feedback state is "**.decode**", no interrupt is pending ("**INT=0**"), and the processor is not being reset (an implied "**RESET=0**"). "**I= ***" indicates that all bits of the instruction register

A macroinstruction, assembly language:

```
11:  ADD #7,R1,R2
```

A macroinstruction, binary code:

```
10:  ...                  [last word of previous instruction]
11:  0110 1000 0010 0001  [first word of instruction]
12:  0000 0000 0000 0111  [immediate value = 7]
13:  ...                  [first word of next instruction]
14:  ...                  [immediate value for next instruction, or
                             first word of instruction after next]
```

The syntax for a source microcode word is:

```
word  <mnemonic>: <inputs>          ::  <outputs>
```

Source microcode for executing the instruction:

```
word DECODE: .decode I= *       INT=0     :: IN->I saveC RJ=> X Y D M RA++->A .get

word #,J,K:  .get    I= 0 1 1             ::         PC++->A  IN=> X          .go

word ADD:    .go     I= * * * 0 1 0 0 0 :: ALUONLY GP= 86 Cin=0 nosh setZNV setC
word ALU->K: .go     I= 0 1 * 0 * * * * :: NOALU   PC++->A    W=> RK       .decode
```

Processor timing in executing the macroinstruction:

micro-cycle number	microcode word(s) being fetched	microcode word(s) controlling processor	memory address being computed	memory address	memory data available	ALU function and bus transfer
	DECODE:	...	PC+1 = 12 (immediate value)	11 (ADD instr.)	...	...
1	#,J,K:	DECODE:	RA+1 (new refresh address)	12 (immediate value)	ADD instr. (latch into I register)	R1=> X,Y,M
2	ADD: and ALU->K:	#,J,K:	PC+1 = 13 (1st word of next instr.)	(refresh address)	7 (immediate value)	7=>X
3	DECODE:	ADD: and ALU->K:	PC+1 = 14 (immed. for next instr.)	13 (1st word of next instr.)	(refresh data)	X+Y->W W=>R2
	...	DECODE:	RA+1 (new refresh address)	14 (immed. for next instr.)	(1st word of next instr.)	...

Figure 5: Example macroinstruction, microcode, and timing

are don't-cares. Previous microcode has ensured that a new macroinstruction was fetched on the previous cycle. Thus "**DECODE:**" latches it into the instruction register ("**IN->I**") at the start of the cycle. The controller has not had time to branch based on the new instruction, but by φ_2 the J and K fields will have arrived at the register decoder; thus this microcode word fetches one of the registers to all of the destinations where it might be needed ("**RJ=> X Y D M**"). This "register prefetch" saves a cycle from most instructions. It is too early to know what to do with the next memory cycle, so the microcode uses it as a refresh cycle ("**RA++->A**", a macro for "**RA->inc Add1 inc->A A->RA**").

The next microcode word "**#,J,K:**" is conditional on the MODE field of the instruction register ("**I= 0 1 1**") and corresponds to an instruction with an immediate value and a register as operands. In the complete microcode, there is also a microcode sequence conditional on each of the other possible values for the MODE, though they are sometimes longer than one cycle, e.g. for memory references. The MODE in this example specifies operand X is an immediate value, which is obtained via the bus from the memory data input buffer ("**IN=> X**"). The PC is incremented past the immediate value ("**PC++->A**") in order to begin fetching the next instruction. The next state output "**.go**" indicates that all operands have been fetched and the code for the operative part of the instruction should take over.

The last two microcode words, "**ADD:**" and "**ALU->K:**" are active simultaneously and complete the macroinstruction. In the "**ADD:**" word the token "**ALUONLY**" indicates that this word specifies only ALU/shifter outputs (i.e. it has no transistors in the OR plane for other outputs) while "**NOALU**" in the "**ALU->K:**" word indicates that this word controls the rest of the outputs. The "**ADD:**" word instructs the ALU/shifter to add its inputs, X and Y, by specifying the appropriate Generate and Propagate codes ("**GP= 66**"), the carry-in ("**Cin=0**"), and the type of shift ("**nosh**", for "no shift"). The complete microcode contains similar words corresponding to the other arithmetic operations: subtract, increment, etc. These words are independent of the MODE field of the instruction but dependent on the OP field (in this example "**I= * * * 0 1 0 0 0**", since the OP code for ADD is **01000**).

The "**ALU->K:**" word deposits the ALU/shifter output in register K ("**W=> RK**"). Other words in the complete microcode, dependent on the MODE but independent of the OP code, handle the other possible destinations. Thus the orthogonality in the macroinstruction set, arithmetic OPs versus MODES, is represented directly in the microcode. Only one microcode word, "**ALU->K:**", is needed to handle two mode cases, since the MODEs have been carefully encoded so that one input condition ("**I= 0 1 * 0**"), decodes both cases. Careful encoding such as this throughout the instruction set helps to keep the microcode compact. In "**ALU->K:**" the PC

is incremented and used as the memory address ("PC++->A"), as it is in the last cycle of all instructions. This begins prefetching the word after the next instruction, in case the next instruction takes an immediate value and needs to use it in its second cycle.

In this example, all three memory cycles are used: instruction fetch, immediate fetch, and refresh cycle. Typical memory cycle usage is perhaps 35% instruction and immediate data fetches, 25% refresh cycles, 10% data reads, 5% data stores, and 25% wasted cycles for discarded prefetches and null reads.

12. Memory

The memory is partitioned into several smaller arrays, as suggested in section 8.5 of [Mead&Conway80]. Each array is 4096 bits, 64 by 64, organized to interface with the processor as 256 16-bit words. The very small amount of read-only memory required for the initialization and bootstrap loader is implemented in a set of "maimed" RAM cells.

The densest read-write memory we understand how to make with MOSIS nMOS technology is based on a 3-transistor dynamic memory cell, which must be refreshed periodically. This refresh function is accomplished by the processor by referencing consecutive memory locations during otherwise unused memory cycles. Commercial single transistor dynamic memories require dynamic node refresh every 2 msec. Systems using such devices typically use error detecting/correcting codes to bring soft errors to acceptable levels. We are depending on the use of 3-transistor cells with fairly large storage nodes, combined with the fast (50 microsecond) refresh rate provided by the processor, to produce sufficiently reliable memory.

The memory cells, figure 6, use separate read-data and write-data busses. This allows simplified control circuitry and shorter cycle time because a read and a write may occur simultaneously. Each memory access starts with a word-line read followed almost always by a refresh write to the same word-line on the next cycle, in parallel with the next read. When a write is requested one of the 4 words read from the selected memory bank is replaced with write data from the processor. This write data is written in the next cycle, in parallel with the next read. (However, if the read is to the same word line as the pipelined write, it accesses stale data which should not be written back on the following cycle. For this reason the refresh write-back is disabled on the second cycle after a write cycle.) In this form of pipelining consecutive writes and write followed by read to the same address will fail. Consecutive writes do not occur in the microcode, and write followed by read to the same address can occur only

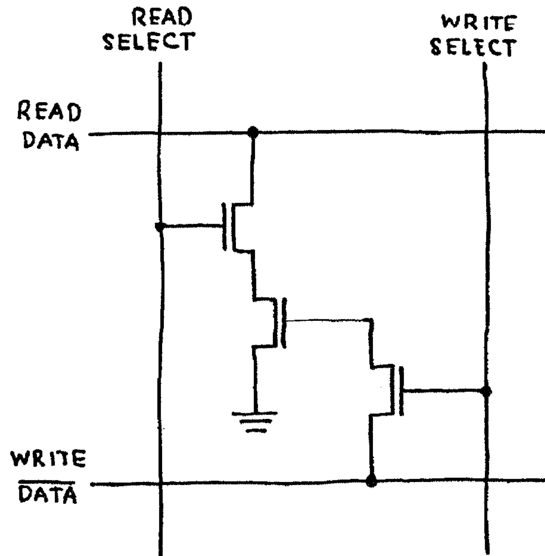


Figure 6: 3-Transistor Dynamic Memory Cell

by writing into the instruction stream.

Steve Rabin is the principal designer for the memory section.

13. Circuit Design

Some of the performance and layout simplicity of Mosaic is due to a "hot clock" design style in which the clock signals may switch between ground and a voltage in excess of Vdd. The simple clock-AND bootstrap driver shown in figure 7 is used extensively and in several variations both in the processor and memory sections. In the memory, the clock-AND is used so extensively that depletion pullup transistors are completely absent.

Although referred to as a "driver", this clock-AND does not provide power amplification of the clock, but rather passes a replica of the hot clock input, whatever its HIGH voltage, to the output as gated by an enable signal of low energy. The clock signal typically switches between ground and 7 volts with Vdd = 5 volts, but the chips also work correctly at reduced power and speed with 5 volt clocks and 5 volt Vdd. The delay and power dissipation of these clock-ANDs is almost negligible, and so the clock driving problem, together with the power dissipation usually required in control signal drivers, is exported to outside the chip where it can be dealt with using special driver circuits.

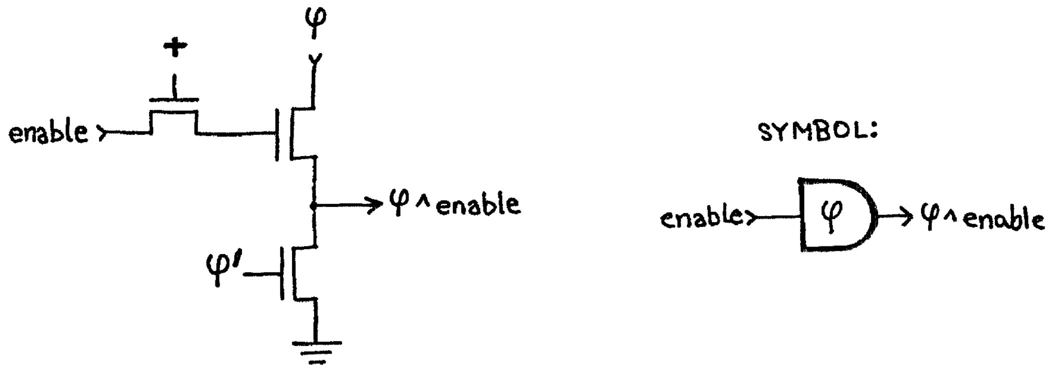


Figure 7: Clock-AND Circuit

This hot clock technique allows pass gates controlled by clock-AND outputs to pass signals with a full 5-volt swing, and makes the chip's performance much less sensitive to variations in the depletion threshold voltage than in conventional Mead-Conway designs.

Precharging is also used extensively in this chip, both to save power and for speed.

Mosaic layout uses Mead-Conway nMOS design rules, substituting buried contacts for butting contacts. Overlaid wires of diffusion, buried contact, and poly are used to produce low-resistance wires which we call "buried wires".

14. Design Tools

The layout and verification was done on a VAX-11/780 running (limping) Berkeley Unix, with design tools written in MAINSAIL and C. Circuit design and optimization relied primarily on tau-model calculations. SPICE was used to evaluate bootstrap effects, technology dependence, and critical timing paths. Extensive SPICE simulations were used to size the ALU carry chain transistors, but the speed improvement over the initial tau-model sizings was only 10% of the carry propagation time, a mere 3% of the processor cycle time.

Cells were laid out initially using colored pencils and graph paper, and then coded in Earl [Kingsley82], a constraint solving geometry and

composition tool. Although the parts are composed in a rectangular bounding box discipline, the geometry internal to cells includes arbitrary angles and approximations of circular arcs, a form of "Boston geometry" that can be specified easily in Earl. This unusual layout style saved about 10% in area over 45-degree angle geometry, and about 25% over Manhattan geometry. The layouts of the ALU, controller, and register array, due to Don Speck, have a visceral appearance characteristic of shameless indulgence in Boston geometry (see appendix C).

For design verification, much of the logic design was coded and simulated using the ternary switch level simulator MOSSIM [Bryant83] to verify logical correctness. After the layout was complete, raster extraction of layout using a Boston geometry circuit extractor produced a switch network that was used in MOSSIM II [Bryant82] simulations.

15. Testing

First silicon, received on 9 February 1983, only 34 days after the CIF was submitted to MOSIS, was tested immediately and found to run code at a 7 MHz clock rate at room temperature. Subsequent processors fabricated using a faster process (still with a 4 micron feature size) ran at up to 11 MHz at room temperature.

A missing contact cut due to a late change was found (missing) before fabricated chips were returned. Subsequent testing revealed two more bugs: an instruction MODE was microcoded incorrectly, and a controller "output type" specified the wrong number of half-cycle delays, causing port read-with-advance to advance before reading. The latter bug escaped detection by the ad hoc simulator because it involved a fractional microcycle phase relationship not represented in the simulator.

Our testing experiences have been quite similar to those reported by several other university groups, and point to two interesting developments in testing for design verification. First, verification tools have become so good that nearly the entire design verification task is now accomplished before first silicon. Second, chips that are systems rather than components turn out to be simpler to test by placing them in their system environment than in a conventional tester.

16. Acknowledgements

Thanks to:

Chuck Seitz for management, design review, Earl coding, and patience

Don Speck for circuit optimization, layout, and verification

Steve Rabin for quality control, verification, and memory section design

Chris Kingsley for Earl

Howard Derby for early design

OM for good ideas

17. References

[Browning80a]
Sally A Browning
Hierarchically Organized Machines
Section 8.4 in [Mead&Conway80]

[Browning80b]
Sally A Browning
The Tree Machine: A Highly Concurrent Computing Environment
Computer Science Technical Report 3760:TR:80, Caltech, 1980

[Browning&Seitz81]
Sally A Browning and C L Seitz
Communication in a Tree Machine
Proc. Second Caltech Conference on VLSI, January 1981
Computer Science, Caltech

[Bryant82]
Randy Bryant, Mike Schuster and Doug Whiting
MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual
Computer Science Technical Report 5033:TR:82, Caltech 1982

[Bryant83]
Randal E Bryant
A Switch-Level Model and Stimulator for MOS Digital Systems
Computer Science Technical Report 5065:TR:83, Caltech, 1983

[Kingsley82]
Chris Kingsley
Earl: An Integrated Circuit Design Language
Computer Science Technical Report 5021:TR:82, Caltech, 1982

[Lutz,Rabin,Seitz&Speck83]
Chris Lutz, Steve Rabin, Chuck Seitz, and Don Speck
Design of the Mosaic Element
Computer Science Technical Report 5093:TR:83, Caltech, 1983
also Proc. MIT Conference on Advanced Research in VLSI, pp. 1-10
Artech Books, 1984

[Mead&Conway80]
Carver A Mead and Lynn Conway
Introduction to VLSI Systems
Addison-Wesley, 1980

[Seitz84]
Charles L Seitz
Experiments with VLSI Ensemble Machines
J. VLSI & CS vol 1, no 3, Computer Science Press, 1984
also Computer Science Technical Report 5102:TR:83, Caltech, 1983

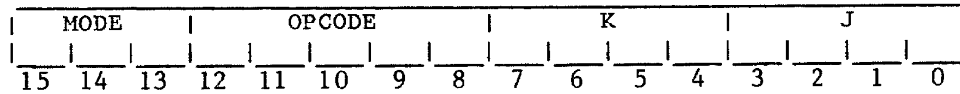
MOSAIC PROCESSOR VERSION A INSTRUCTION SET   page 1 of 3

(1-JUN-83)

PROCESSOR FEATURES:

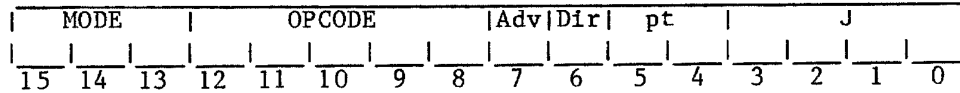
Sixteen 16-bit general registers: R0 ... R15
Memory addressed as 16-bit words
12-bit Program Counter (PC) contains address of next instruction
Flags: C -- Carry/Not Borrow
Z -- Zero (all 16 result bits zero)
N -- Negative (bit 15 of result)
V -- Twos-complement overflow
Ports: Four input ports
Four output ports
Connecting an input port to an output port forms a fifo two 16-bit words long.

ALL INSTRUCTIONS:



followed optionally by one word of immediate value.

When K specifies a port:



pt is the port number; specifies one of 4 ports
Dir=0 for output port; Dir=1 for input port
Adv=1 to advance port (remove word from fifo) after input port is read

KEY: Rn is register number n.
val is the an immediate value.
@z is the memory word whose address is z.
A | B is the concatenation of bit field A and bit field B .
f<i> means i-th bit of f .
f<i:j> means i-th to j-th bits of f .
Bits are numbered from least to most significant.
outport is output port number pt.
inport is input port number pt. If Adv=1 then the port is advanced after reading its value.

SPECIAL CASES: RESET: (Reset pin goes HIGH)
C|V|N|Z|PC -> @(-1); 0 -> PC

INSTRUCTION MODES:

All instructions fetch operands X and Y as specified by MODE, then perform the operation specified by OPCODE.

MODE	Dir	X	Y	Dest	Assembly language syntax
0		RJ	RK	RK	<Mnemonic> Rj {, Rk}
1		val	RJ	RK	<Mnemonic> #val , Rj {, Rk}
2		@RJ	RK	RK	<Mnemonic> @Rj {, Rk}
3		@val	RJ	RK	<Mnemonic> @#val , Rj {, Rk}
4	0	RJ	0	outport	<Mnemonic> Rj , Ppt
5	0	val	RJ	outport	<Mnemonic> #val {, Rj} , Ppt
4	1	inport	RJ	RJ	<Mnemonic> <input> , Rj
5	1	val	inport	RJ	<Mnemonic> #val , <input> , Rj
6		@RJ	RK	@RJ	<Mnemonic> M @Rj {, Rk}
7		@val	RJ	@val	<Mnemonic> M @#val {, Rj}

<input> ::= Ppt= to read input port pt without advancing
Ppt+ to read input port pt, then advance port

Note: Modes 6 and 7 are defined only for instructions that assign a result to Dest.

ARITHMETIC INSTRUCTIONS:

Arithmetic Instructions are those which assign a result to Dest.
All arithmetic instructions modify the Z, N, and V flags.
(Some instructions always set V to 0.
They are MOV, COM, RNR, RNR, ASR, LSR, AND, OR, and XOR.)

<table>
<tr><th>OPCODE</th><th>INSTRUCTION</th><th><Mnemonic></th><th>EFFECT</th><th></th><th>CARRY FLAG MODIFIED?</th></tr>
<tr><td>00</td><td>MOVe</td><td>MOV</td><td>X</td><td>-> Dest</td><td>no</td></tr>
<tr><td>01</td><td>bitwise COMplement</td><td>COM</td><td>~X</td><td>-> Dest</td><td>no</td></tr>
<tr><td>02</td><td>INCrement</td><td>INC</td><td>X + 1</td><td>-> Dest</td><td>no</td></tr>
<tr><td>03</td><td>DECrement</td><td>DEC</td><td>X - 1</td><td>-> Dest</td><td>no</td></tr>
<tr><td>04</td><td>NEGate</td><td>NEG</td><td>-X</td><td>-> Dest</td><td>yes</td></tr>
<tr><td>05</td><td>Rotate Nibble Right</td><td>RNR</td><td>X<3:0> | X<15:4></td><td>-> Dest</td><td>no</td></tr>
<tr><td>06</td><td>ROtate Right</td><td>ROR</td><td>C|X</td><td>-> Dest|C</td><td>yes</td></tr>
<tr><td>07</td><td>ROtate Left</td><td>ROL</td><td>X + X + C</td><td>-> Dest</td><td>yes</td></tr>
<tr><td>08</td><td>Arithmetic Shift Right</td><td>ASR</td><td>X<15>|X</td><td>-> Dest|C</td><td>yes</td></tr>
<tr><td>09</td><td>Logical Shift Right</td><td>LSR</td><td>0|X</td><td>-> Dest|C</td><td>yes</td></tr>
<tr><td>0A</td><td>ADD</td><td>ADD</td><td>X + Y</td><td>-> Dest</td><td>yes</td></tr>
<tr><td>0B</td><td>ADD with Carry</td><td>ADDC</td><td>X + Y + C</td><td>-> Dest</td><td>yes</td></tr>
<tr><td>0C</td><td>SUBtract</td><td>SUB</td><td>Y - X</td><td>-> Dest</td><td>yes</td></tr>
<tr><td>0D</td><td>bitwise AND</td><td>AND</td><td>X and Y</td><td>-> Dest</td><td>no</td></tr>
<tr><td>0E</td><td>bitwise OR</td><td>OR</td><td>X or Y</td><td>-> Dest</td><td>no</td></tr>
<tr><td>0F</td><td>bitwise eXclusive OR</td><td>XOR</td><td>X exclusive or Y</td><td>-> Dest</td><td>no</td></tr>
</table>

MOSAIC PROCESSOR VERSION A INSTRUCTION SET

NON-ARITHMETIC INSTRUCTIONS:

OPCODE	INSTRUCTION	<Mnemonic>	EFFECT
10	CoMPare	CMP	modify Z,N,V,C based on X-Y
11	JUMP	JUMP	X<11:0> -> PC
12	undefined		
13	undefined		
14	PUSH	PUSH	Y-1 -> RK; X -> @RK; modify Z,N,V based on X
15	STOre X at y	STOX	X -> @Y; modify Z,N,V based on X
16	undefined		
17	undefined		
18	PUSH Jump	PUSHJ	Y-1 -> RK; C|V|N|Z|PC -> @RK; X<11:0> -> PC
19	STOre Y at x	STOY	Y -> @X; modify Z,N,V based on Y
1A	POP	POP	@RK -> RJ; RK+1 -> RK; modify Z,N,V based on RJ
1B	POP Jump	POPJ	@RK -> PC; RK+1 -> RK
1C	BRAnch True	BRAT	If Condition k is True then X<11:0> -> PC
1D	BRAnch False	BRAF	If Condition k is False then X<11:0> -> PC
1E	undefined		
1F	undefined		

BRANCH CONDITIONS:

K	Condition		Alternate <Mnemonic> (implies OPCODE=BRAT)	(implies OPCODE=BRAF)
00pt	Output port number pt Not Ready (i.e. no room in port to do output)		BONR	BOR
01pt	Input port number pt Not Ready (i.e. no word in input port to read)		BINR	BIR
1000	V	[overflow]	BVS	BVC
1001	N	[negative]	BNS	BNC
1010	~C	[Carry = 0]	BCC or BLO	BCS or BHIS
1011	N xor V	[signed <]	BLT	BGE
1100	Z	[zero]	BZS or BEQ	BZC or BNE
1101	Z or N	[<= zero]	BLEZ	BGTZ
1110	Z or ~C	[unsigned <=]	BLOS	BHI
1111	Z or (N xor V)	[signed <=]	BLE	BGT

When <OP Mnemonic> is BONR, BOR, BINR, or BIR:
pt may be specified by writing "Ppt" in place of "Rk" field.

```
! Mosaic ver. A microcode    page 1 of 3


  Version  14-Mar-83

! MACROS DEFINED IN ASSEMBLER:
! Cin=0      ::  Cforce
! Cin=1      ::  Cforce Cvall
! PC++->A    ::  PC->inc Addl inc->A A->PC
! RA++->A    ::  RA->inc Addl inc->A A->RA
! PC->A      ::  PC->inc inc->A A->PC
! X=>        ::  GP= OC Cin=0 nosh W=>
! Y=>        ::  GP= OA Cin=0 nosh W=>
! RJ=>       ::  useJ R=>
! RK=>       ::  R=>
! RJ         ::  useJ R
! RK         ::  R
! Y+1=>      ::  GP= OA Cin=1 nosh W=>
! Y-1=>      ::  GP= A5 Cin=0 nosh W=>
! saveC      ::  rnib
! TESTX      ::  GP= OC Cin=0 nosh setZNV
!      All alu outputs no xistors:
! NOALU      ::  GP= FF cforce setC saveC lsr ror asr rnib nosh
!      Xistors only in alu outputs:
! ALUONLY    ::  RA->inc PC->inc IN->I Advance Pt=> RJ=> RK=> Pt RJ RK .FbackF

! Feedback mnemonics begin with a '.' .
! Word names end with a ':' .

! 'I=' starts instr register mask starting with I<15>.
! Unspecified bits are '*'.
! When 'I=' not given in some row, that from last row is used.

! 'RESET=0' implied when 'RESET-1' not specified.

! 'Word0' initiates outputs used when I<8>=0.
! 'Word1' initiates outputs used when I<8>=1.
! 'Word' initiates outputs used independent of I<8>.
```

```
! Mosaic ver. A microcode    page 2 of 3


row RESET=1  I= * * * * * * * * * * *  Word RESET: saveC  PC->A PC=> D .reset2
      ! Take FFFF off the precharged undriven bus as old PC destination.
row .reset2 Word RESET2:   A  A->PC  A->RA                          .reset3
      ! Take first instruction from location 0 (=FFFF+1)
row .reset3 Word RESET3: Write  PC++->A                             .fetch


row .fetch     Word FETCH:  PC++->A  .decode


! All instructions begin with DECODE
row .decode    Word DECODE:    IN->I             RJ=> X Y D  RA++->A  .get


! Fetch operands for all instrs:  Word name is "<First op>,<Second op>,<Dest>"
row .get  I= 0 0 0  Word J,K,K:    saveC  PC->A  RK=> Y            .go
row .get  I= 0 0 1  Word #,J,K:    saveC  PC++->A  IN=> X D        .go

row .get  I= * 1 0  Word @J,K:     saveC  RJ=> A                   .get3
row .get3 I= 0 1 0  Word @J,K,K:           PC->A  RK=> Y           .get4
row .get3 I= 1 1 0  Word @J,K,@J:                 RK=> Y           .get4

row .get  I= * 1 1  Word @#,J:     saveC  PC++->A  IN=> X          .get2
row .get2           Word @#,J2:                    X=> A           .get3
row .get3 I= 0 1 1  Word @#,J,K:           PC->A                   .get4
row .get3 I= 1 1 1  Word @#,J,@#:                                  .get4


row .get4 I= * 1 *  Word any@:           IN=> X D                  .go

row .get  I= 1 0 0  Word io;no#:   saveC  PC->A                    .io
row .get  I= 1 0 1  Word io;with#: saveC PC++->A  IN=> X Y D       .io

row .io   I= 1 0 *   PortC=1 Word I/O wait:  X=> X Y D RA++->A  .get2
row .get2 I= 1 0 *          Word wait2:      PC->A       saveC  .io

row .io   I= 1 0 * * * * * * * * 0 PortC=0 Word Willsend: RJ=> Y          .go
row .io   I= 1 0 0 * * * * * 0 1 PortC=0 Word Get:     Pt=> X D           .go
row .io   I= 1 0 1 * * * * * 0 1 PortC=0 Word Get:     Pt=> Y             .go
row .io   I= 1 0 0 * * * * * 1 1 PortC=0 Word GetAdv: Pt=> X D Advance .go
row .io   I= 1 0 1 * * * * * 1 1 PortC=0 Word GetAdv: Pt=> Y   Advance .go


! Arithmetic instructions.  2 rows active simultaneously.
! Rows to handle results:
row .go  I= 0 * * 0             Word Alu->K:  PC++->A NOALU  W=> RK  .decode
row .go  I= 1 0 * 0 * * * * * 1 Word Alu->J:  PC++->A NOALU  W=> RJ  .decode
row .go  I= 1 0 * 0 * * * * * 0 Word Alu->Out: PC++->A NOALU  W=> Pt .decode
row .go  I= 1 1 * 0             Word Alu->M:          NOALU  W=> D   .sto
row .sto I= * * * *             Word Alu->M2:  Write  PC->A          .fetch
```

```
! Mosaic ver. A microcode    page 3 of 3


! Rows to specify ALU function:
row .go  I= * * * 0 0 0 0 Word0 mov:  ALUONLY GP= 0C Cin=0 nosh setZNV
                          Word1 com:  ALUONLY GP= 03 Cin=0 nosh setZNV
row .go  I= * * * 0 0 0 1 Word0 inc:  ALUONLY GP= 0C Cin=1 nosh setZNV
                          Word1 dec:  ALUONLY GP= C3 Cin=0 nosh setZNV
row .go  I= * * * 0 0 1 0 Word0 neg:  ALUONLY GP= 03 Cin=1 nosh setZNV setC
                          Word1 rnib: ALUONLY GP= 0C Cin=0 rnib setZNV
row .go  I= * * * 0 0 1 1 Word0 ror:  ALUONLY GP= 0C Cin=0 ror  setZNV
                          Word1 rol:  ALUONLY GP= C0       nosh setZNV setC
row .go  I= * * * 0 1 0 0 Word0 asr:  ALUONLY GP= 0C Cin=0 asr  setZNV
                          Word1 lsr:  ALUONLY GP= 0C Cin=0 lsr  setZNV
row .go  I= * * * 0 1 0 1 Word0 add:  ALUONLY GP= 86 Cin=0 nosh setZNV setC
                          Word1 addc: ALUONLY GP= 86       nosh setZNV setC
row .go  I= * * * 0 1 1 0 Word0 sub:  ALUONLY GP= 29 Cin=1 nosh setZNV setC
                          Word1 and:  ALUONLY GP= 08 Cin=0 nosh setZNV
row .go  I= * * * 0 1 1 1 Word0 or:   ALUONLY GP= 0E Cin=0 nosh setZNV
                          Word1 xor:  ALUONLY GP= 06 Cin=0 nosh setZNV


! COMPARE (set flags based on X-Y) and JUMP
row .go I= * * * 1 0 0 * Word0 cmp:  PC++->A GP= 49 Cin=1 nosh setZNV setC
                                                                         .decode
                         Word1 jump:   X=> A  A->PC                   .fetch

! Pushes and explicit stores
row .go I= * * * 1 0 1 * Word0 push:   Y-1=> RK A             .go2
                         Word1 stoX:   Y=> A                  .go2
row .go2                 Word0 push2:  Write  PC->A  testX    .fetch
                         Word1 stoX2:  Write  PC->A  testX    .fetch

row .go I= * * * 1 1 0 0 Word0 pushj:  PC=> D                 .go2
                         Word1 stoY:   Y=> D  setZNV          .go2
row .go2                 Word0 pushj2: Y-1=> RK A             .go3
                         Word1 stoY2:  X=> A                  .go3
row .go3                 Word0 pushj3: Write X=> A  A->PC     .fetch
                         Word1 stoY3:  Write PC->A            .fetch

! Pops
row .go I= * * * 1 1 0 1  Word  pop(j):   RK=> Y A            .go2
row .go2                  Word  pop(j)2:  PC->A  Y+1=> RK     .go3
row .go3                  Word0 pop3:    PC++->A  IN=> RJ X   .decode
                          Word1 popj3:  IN=> A  A->PC         .fetch

! Conditional Branches
row .go I= * * * 1 1 1 * * 0  PortC=0 Word0 BraT_Pt=0:   PC++->A      .decode
                                      Word1 BraF_Pt=0:   X=> A A->PC  .fetch
row .go I= * * * 1 1 1 * * 0  PortC=1 Word0 BraT_Pt=1:   X=> A A->PC  .fetch
                                      Word1 BraF_Pt=1:   PC++->A      .decode

row .go I= * * * 1 1 1 * * 1  FlagC=0 Word0 BraT_Fl=0:   PC++->A      .decode
                                      Word1 BraF_Fl=0:   X=> A A->PC  .fetch
row .go I= * * * 1 1 1 * * 1  FlagC=1 Word0 BraT_Fl=1:   X=> A A->PC  .fetch
                                      Word1 BraF_Fl=1:   PC++->A      .decode
```

MOSAIC VERSION A CIRCUIT DIAGRAMS 4-84

Pads (46):

Ground	Reset	Address Pad⟨Ø...11⟩	scan_in Pad
+Vdd	Write	Data Pad ⟨Ø...15⟩	scan_out Pad
φ_1	φ_a	IP Pad Ø...3	
φ_2	φ_b	OP Pad Ø...3	

Controller Inputs (17):

Reset	I⟨6...15⟩
FCond	FB⟨Ø...3⟩ (feedback)
PCond	

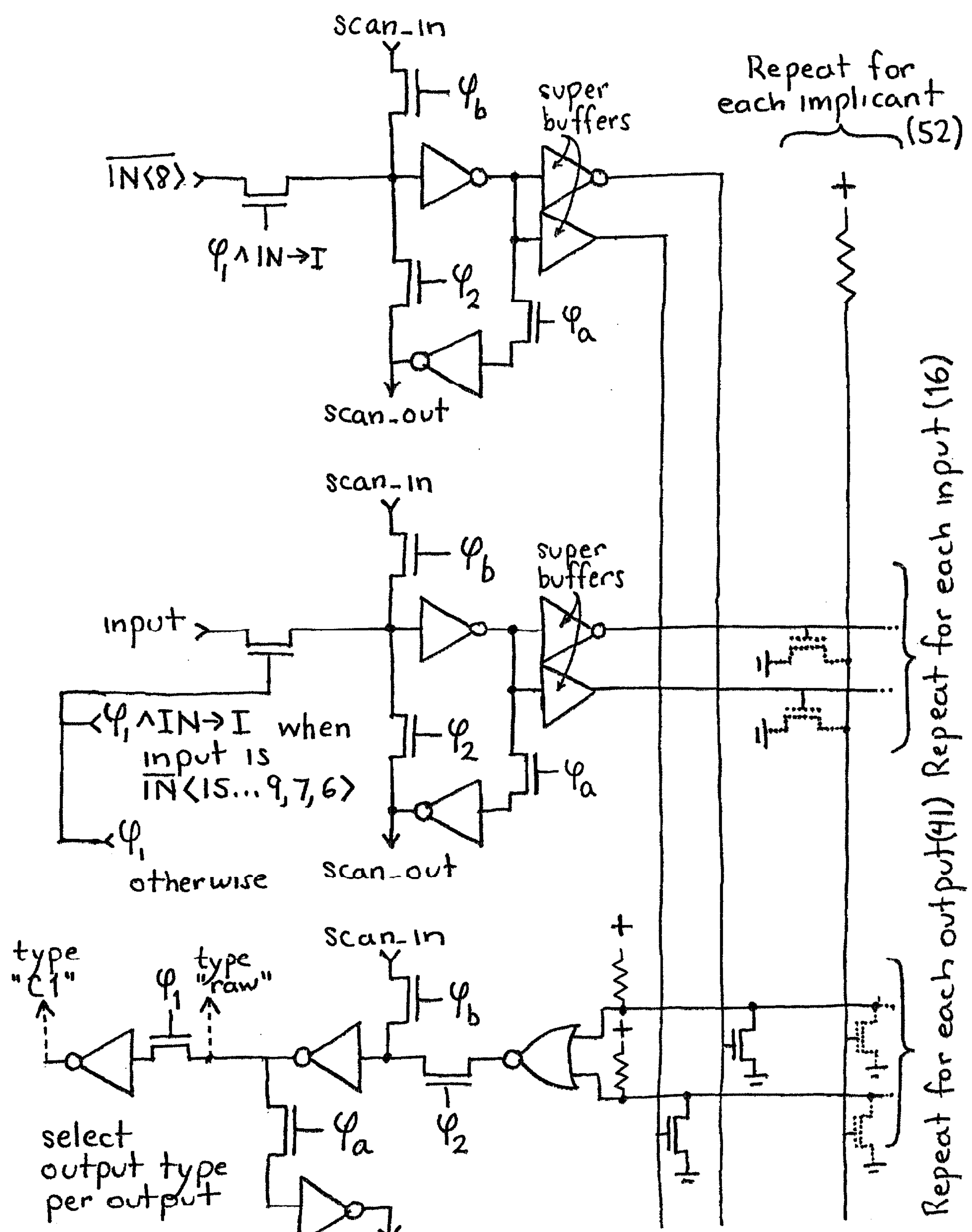
Controller Outputs (41):

* indicates type "raw". All others type "C1".

Bus sources:	W⇒	PC⇒	IN⇒	$\overline{\text{R⇒}}$	$\overline{\text{Pt⇒}}$
Bus destinations:	⇒X	⇒Y	⇒D	$\overline{\text{⇒R}}$	$\overline{\text{⇒Pt}}$
	⇒A	[⇒F ver. B only]			
ALU/shifter:	$P_{ØØ}$*	$P_{Ø1}$*	$P_{1Ø}$*	P_{11}*	
		$G_{Ø1}$*	$G_{1Ø}$*	G_{11}*	
	rnib*	ror*	lsr*	arr*	nosh*
	setC*	setZNV	Cforce*	$\overline{\text{Cval}}$*	
Address section:	A→RA	A→PC	RA→inc*	PC→inc*	inc→A Add1
Misc.:	$\overline{\text{advance}}$	write	IN→I	useJ	FB⟨Ø...3⟩

Controller

Mosaic ver. A 4-84



Connect scan_in's to scan_out's.
First scan_in to scan_in Pad.
Last scan_out to scan_out Pad.

ALU

Mosaic ver.A 4-84

Repeat for i= Ø to 15:

$\varphi_2 \wedge \Rightarrow X$

X<i>

$\varphi_2 \wedge \Rightarrow Y$

Y<i>

Propagate function block

P<i>

Generate function block

G<i>

Bus <i>

$\varphi_1 \wedge P_{00}$ $\varphi_1 \wedge P_{01}$ $\varphi_1 \wedge P_{10}$ $\varphi_1 \wedge P_{11}$

$\parallel (G_{00})$ $\varphi_1 \wedge G_{01}$ $\varphi_1 \wedge G_{10}$ $\varphi_1 \wedge G_{11}$

$\overline{\text{aluC<i+1>}}$

φ_2

G<i>

$\overline{\text{aluout<i>}}$

P<i>

restoring (i=0,3,6,9,12,15)

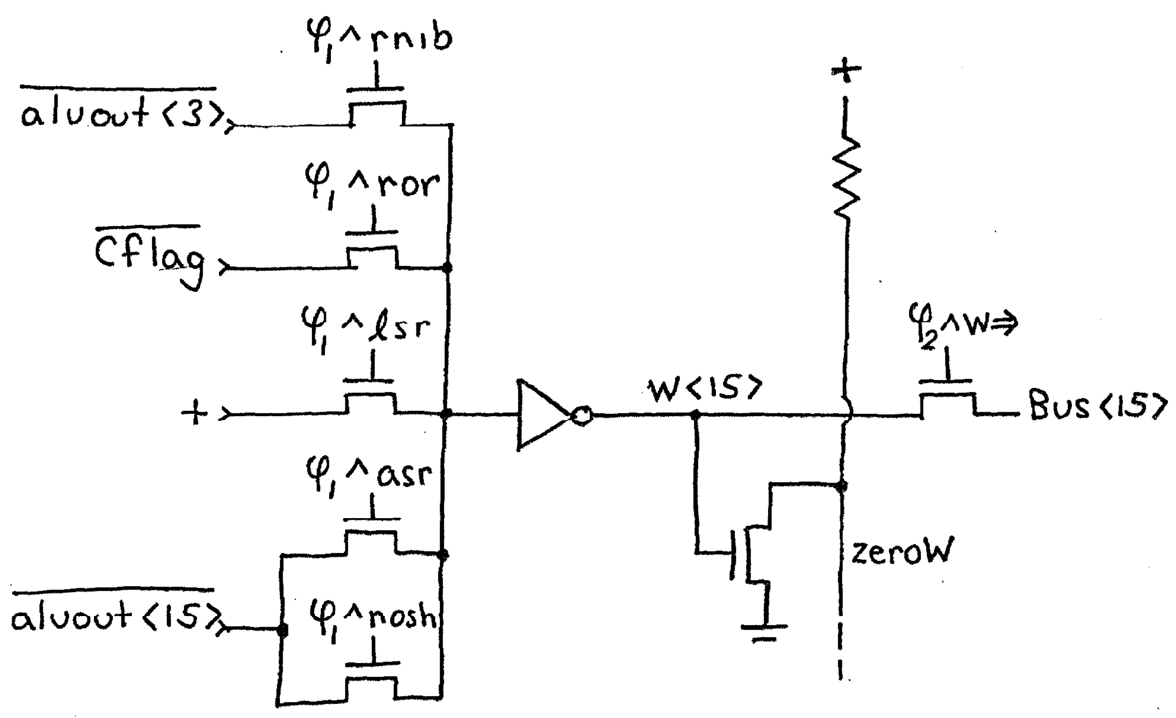
not restoring (all other i)

$\overline{\text{aluC<i>}}$

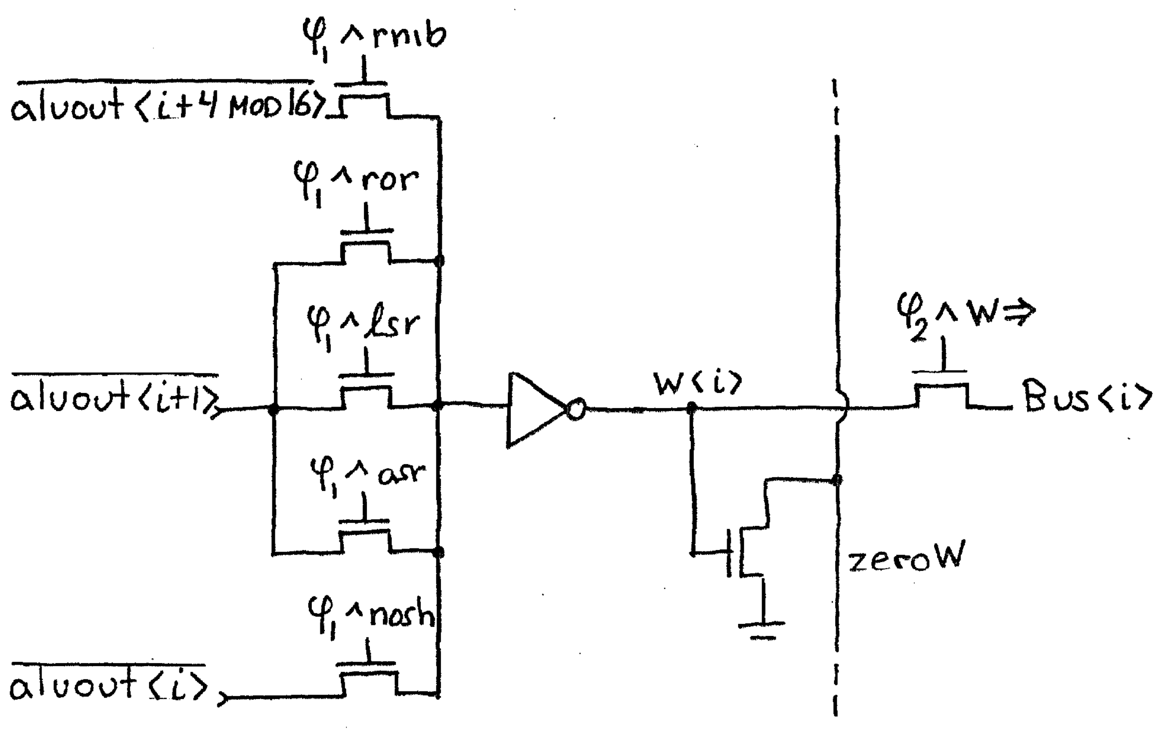
Shifter

Mosaic ver. A 4-84

Bit 15:

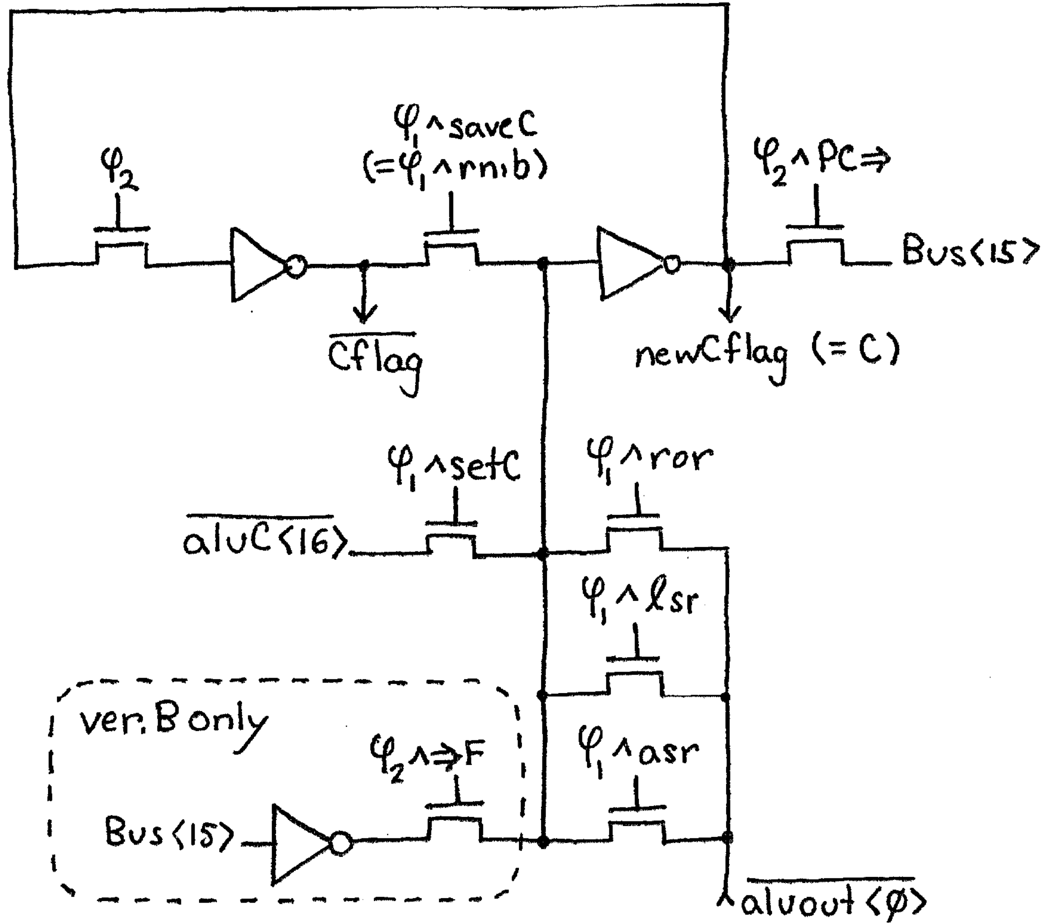


Repeat for i = Ø to 14:

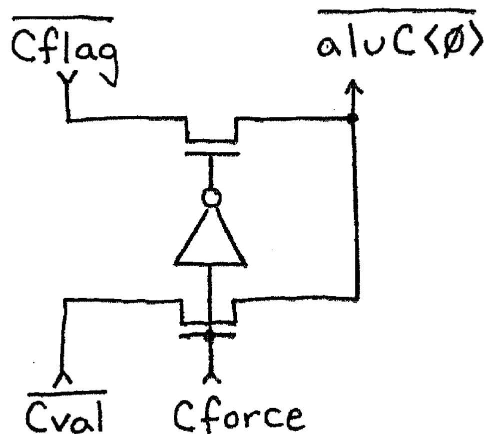


Carry flag (and ALU Carry In)

Mosaic ver. A 4-84

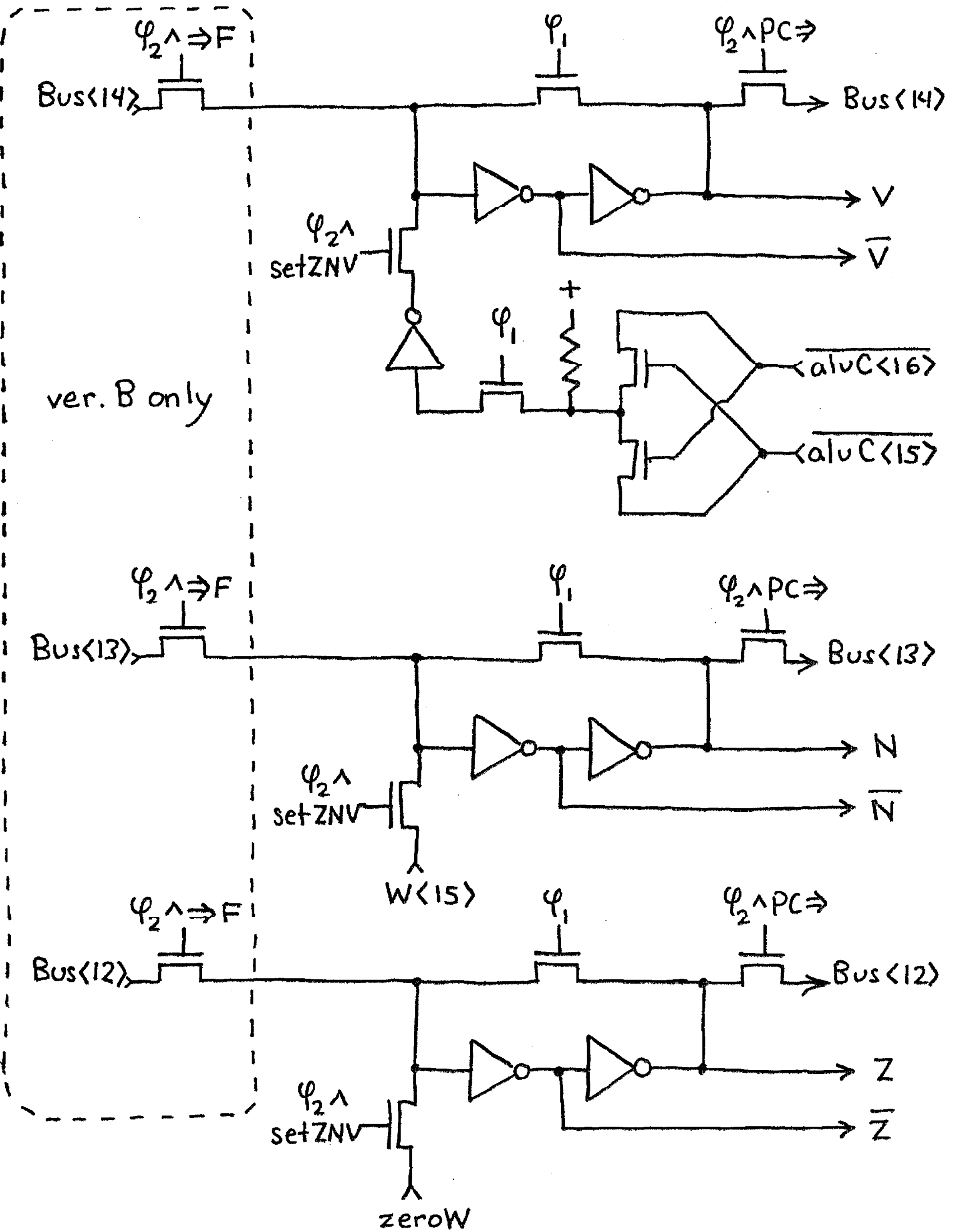


ALU Carry in



V, N, and Z flags

Mosaic ver. A 4-84



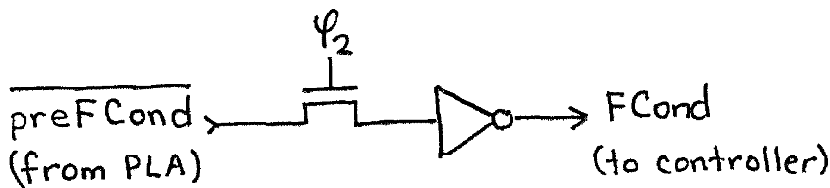
Flag Condition PLA

Mosaic ver. A 4-84

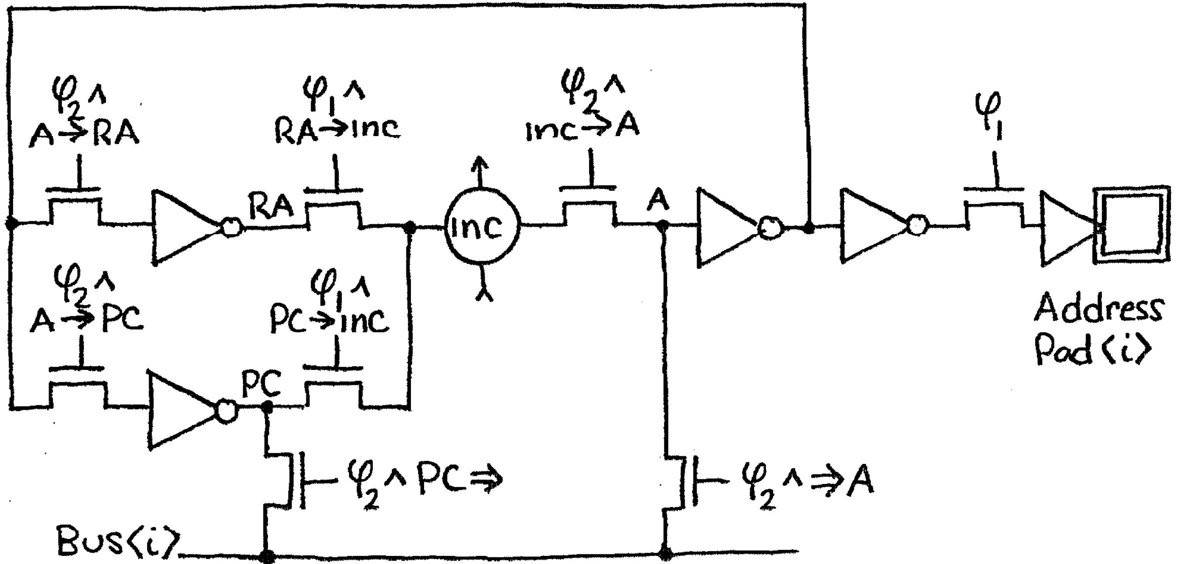
I⟨6⟩	I⟨5⟩	I⟨4⟩	preFCond	
Ø	Ø	Ø	V	
Ø	Ø	1	N	[< Ø]
Ø	1	Ø	$\overline{C}$	[unsigned <]
Ø	1	1	$N \oplus V$	[signed <]
1	Ø	Ø	Z	[= Ø]
1	Ø	1	$Z \vee N$	[≤ Ø]
1	1	Ø	$Z \vee \overline{C}$	[unsigned ≤]
1	1	1	$Z \vee (N \oplus V)$	[signed ≤]

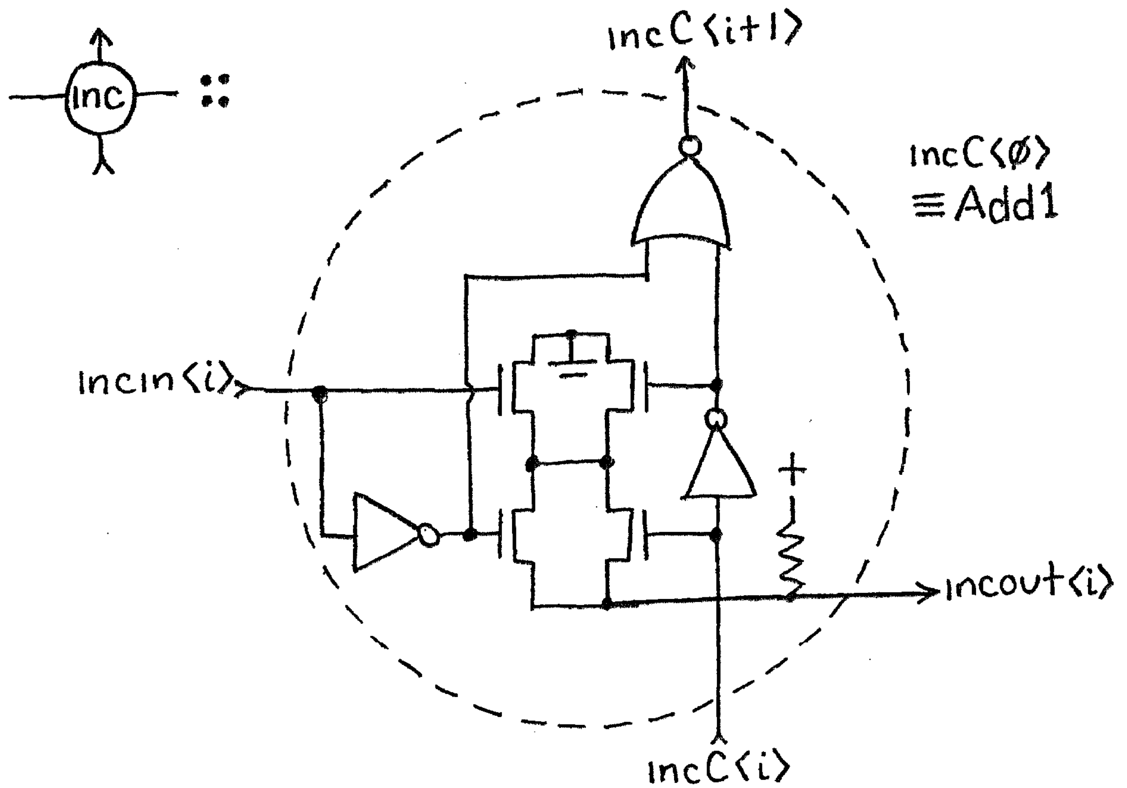
$$
\begin{aligned}
preFCond = \; & (I\langle 6\rangle \wedge Z) \\
& \vee (\overline{I\langle 6\rangle} \wedge \overline{I\langle 5\rangle} \wedge \overline{I\langle 4\rangle} \wedge V) \\
& \vee (\overline{I\langle 5\rangle} \wedge I\langle 4\rangle \wedge N) \\
& \vee (I\langle 5\rangle \wedge \overline{I\langle 4\rangle} \wedge \overline{C}) \\
& \vee (I\langle 5\rangle \wedge I\langle 4\rangle \wedge N \wedge \overline{V}) \\
& \vee (I\langle 5\rangle \wedge I\langle 4\rangle \wedge \overline{N} \wedge V)
\end{aligned}
$$

Use PLA to compute NOR-NOR form of $\overline{preFCond}$



Repeat for i = Ø to 11:

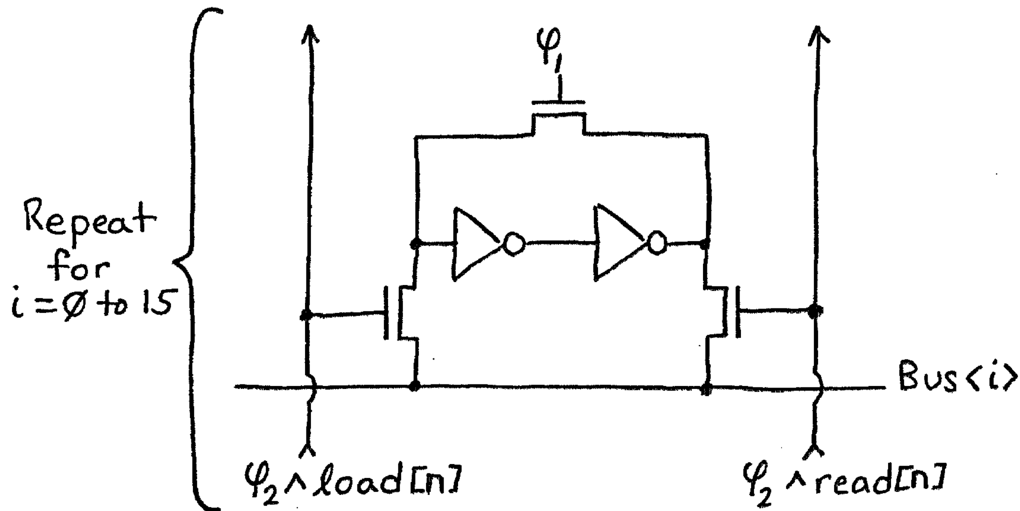


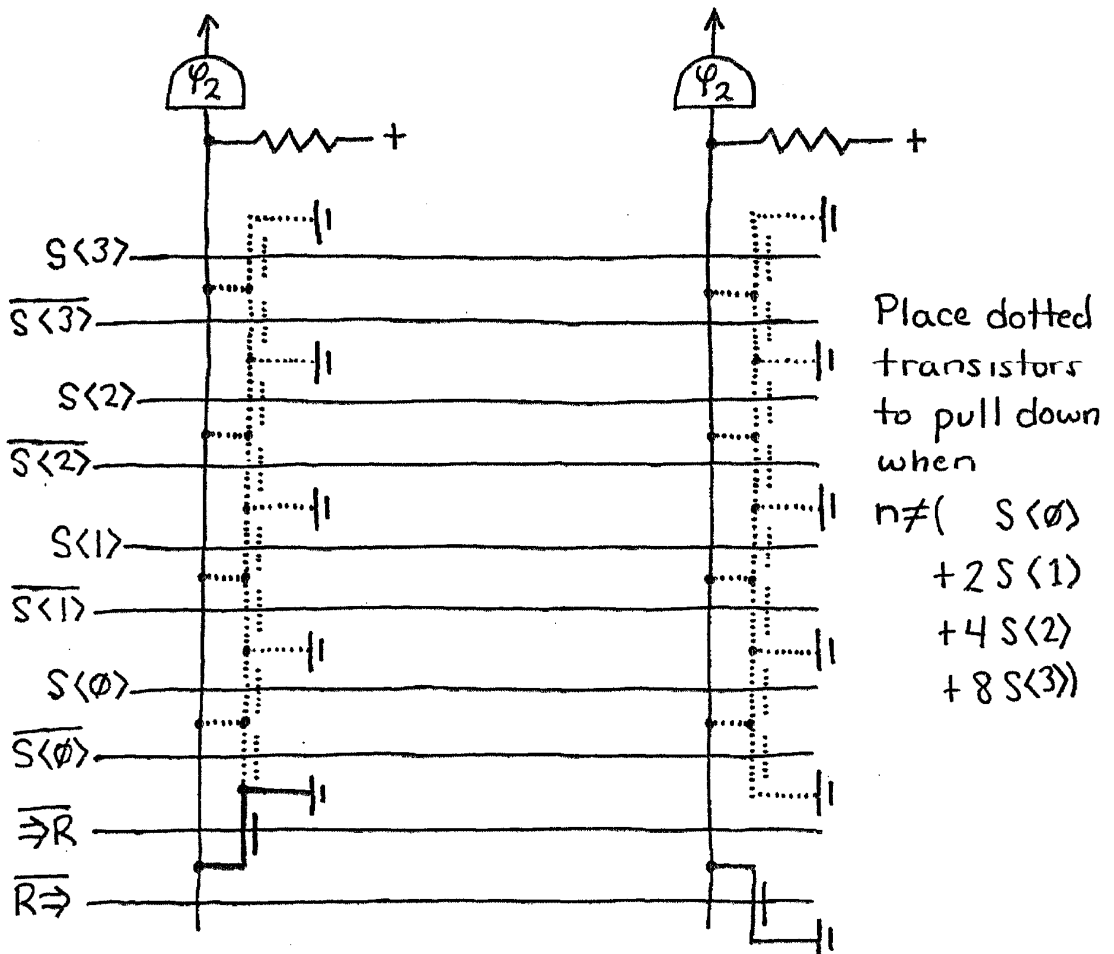


Register Array

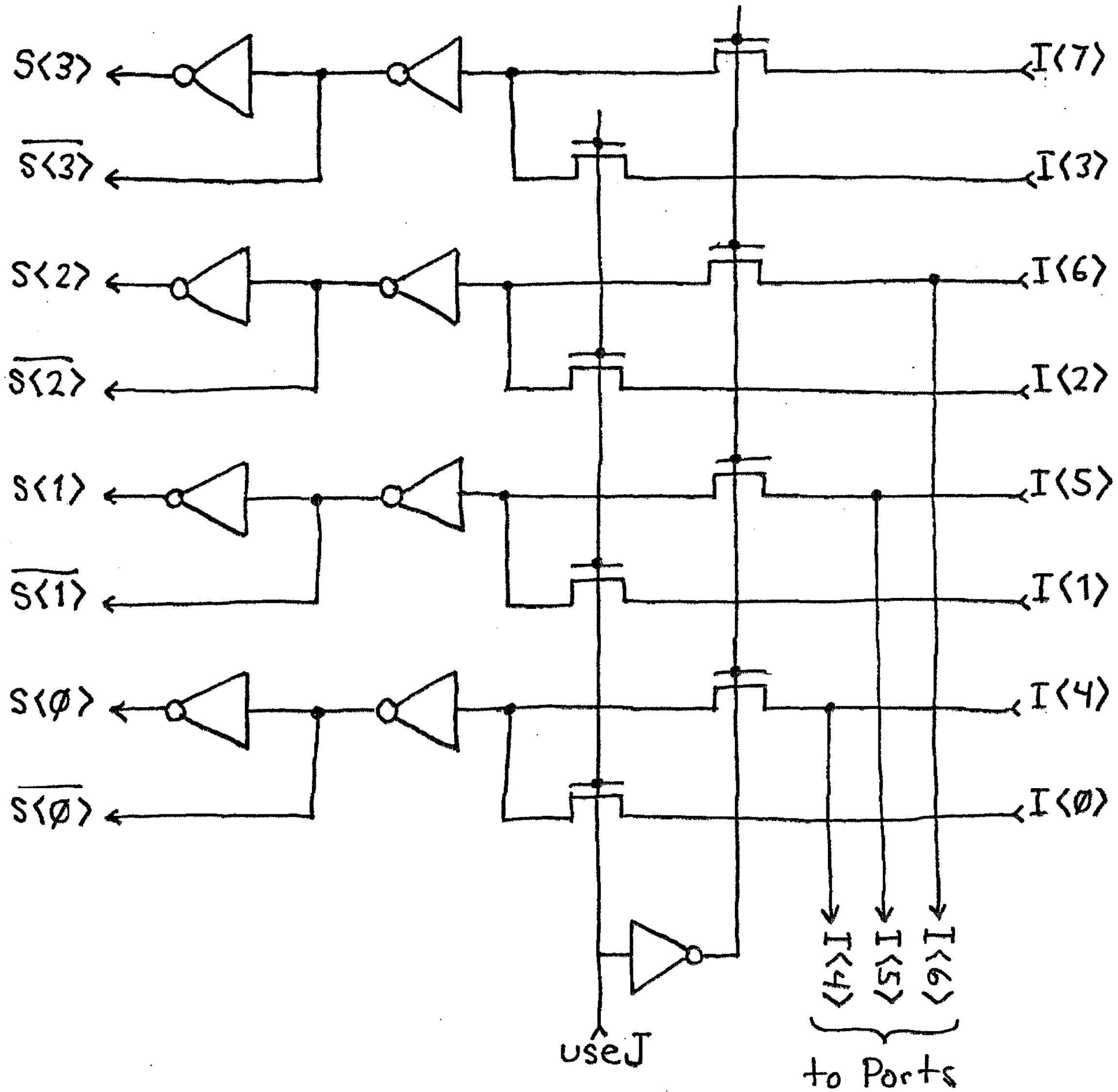
Mosaic ver. A 4-84

Repeat for n = Ø to 15:





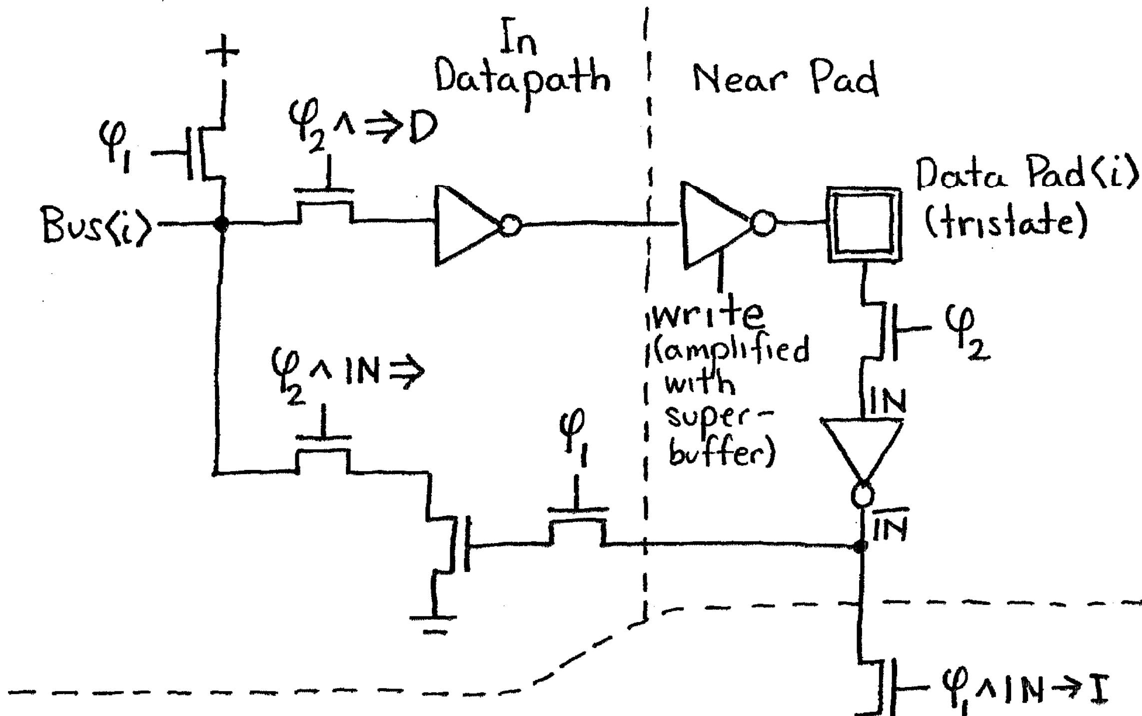
S (Register Select) Generation   Mosaic ver. A 4-84



Mosaic ver. A 4-84

Memory Data Interface, Bus Precharge and Instruction Register

Repeat for $i = \emptyset$ to 15:

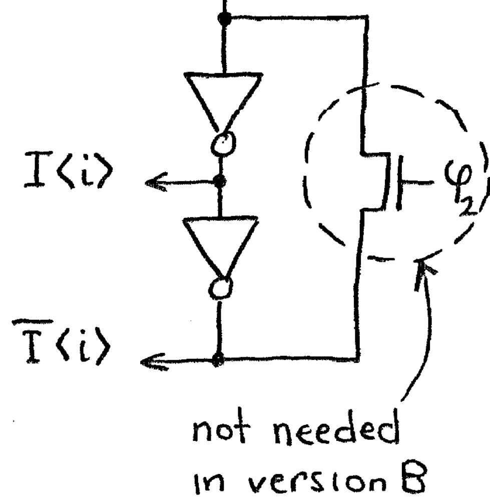


For $i = 4, 5, 6$:
near flag condition PLA

For $i = \emptyset \ldots 7$:
near register select

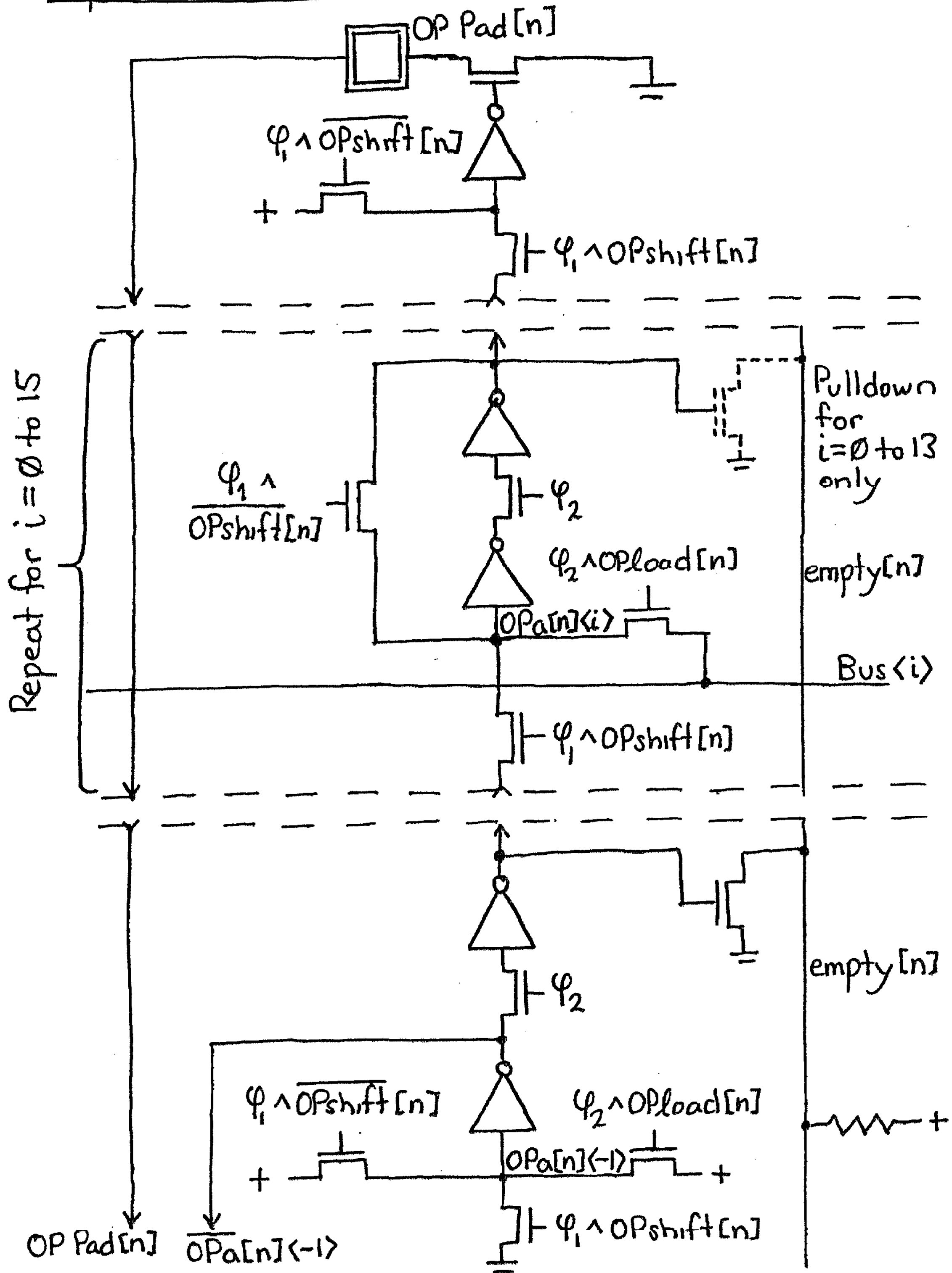
For $i = 6 \ldots 15$:
equivalent circuit built into controller

(Note duplications for some bits)



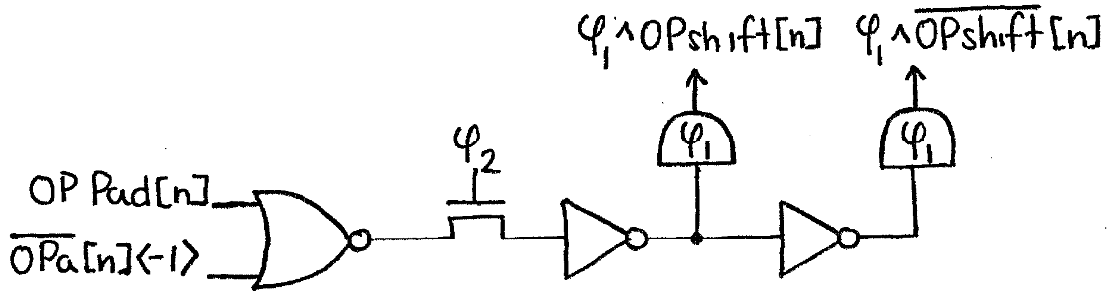
Output Ports pg. 1 of 2

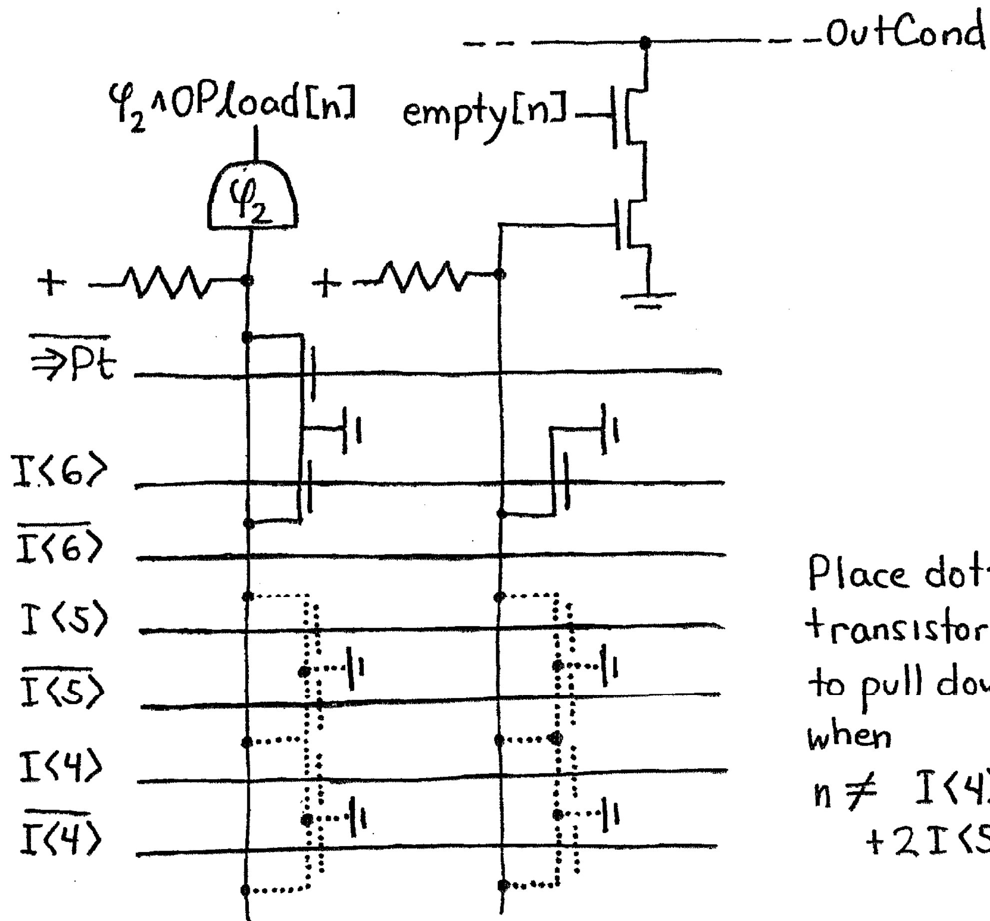
Repeat for n = Ø to 3:



Output Ports pg. 2 of 2 Mosaic ver. A 4-84

Repeat for n = Ø to 3:



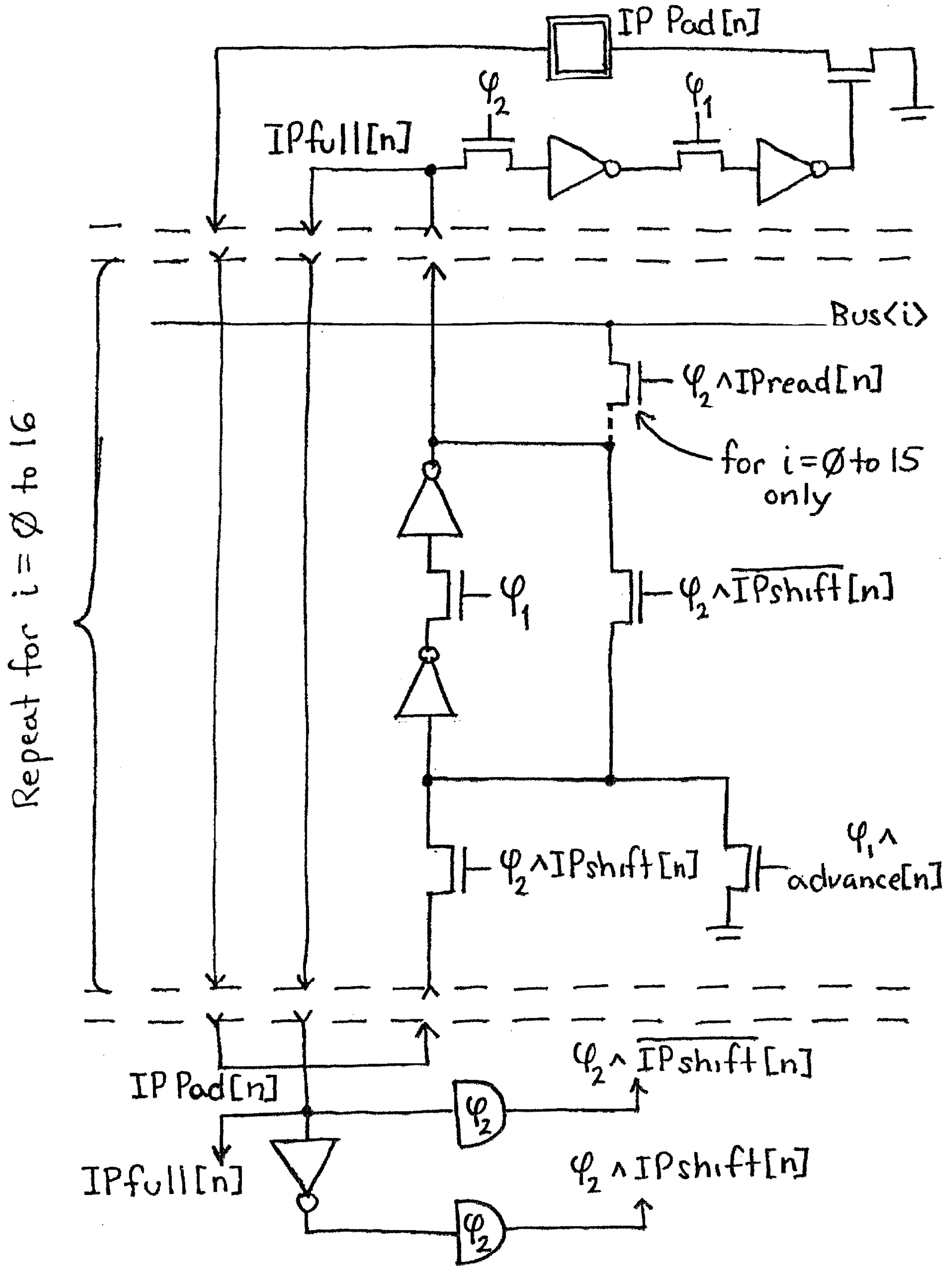


Place dotted transistors to pull down when n ≠ I<4> + 2 I<5>

Input Ports pg. 1 of 2

Mosaic ver. A 4-84

Repeat for n = Ø to 4:



Input Ports pg. 2 of 2 (and Port Condition) Mosaic ver. A 4-84

Repeat for = Ø to 3:

$\varphi_1 \wedge$ advance[n]

φ_1

prePCond

$\varphi_2 \wedge$ IPread[n]

IPfull[n]

φ_2

φ_2

+ + +

$\overline{Pt\Rightarrow}$

$\overline{advance}$

I⟨6⟩

$\overline{I\langle 6\rangle}$

I⟨5⟩

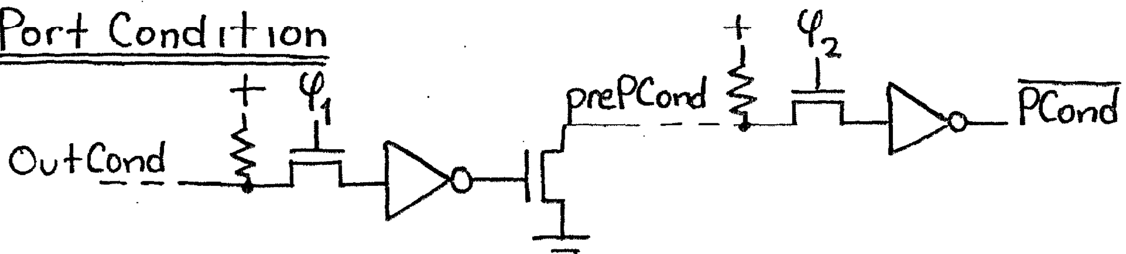
$\overline{I\langle 5\rangle}$

I⟨4⟩

$\overline{I\langle 4\rangle}$

Place dotted transistors to pull down when n ≠ I⟨4⟩ +2I⟨5⟩

Port Condition



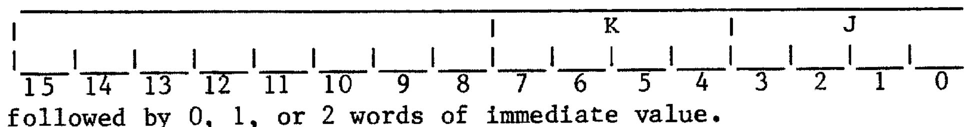
MOSAIC PROCESSOR VERSION B INSTRUCTION SET   page 1 of 4

(6-MAR-84)

PROCESSOR FEATURES:

```
    Sixteen 16-bit general registers:  R0 ... R15
    Memory addressed as 16-bit words
    12-bit Program Counter (PC) contains address of next instruction
    Flags:  C -- Carry/Not Borrow
            Z -- Zero
            N -- Negative
            V -- Twos-complement overflow
    Ports:  Four input ports
            Four output ports
            Connecting an input port to an output port forms a fifo
                two 16-bit words long.
```

ALL INSTRUCTIONS:



followed by 0, 1, or 2 words of immediate value.

When K specifies a port:

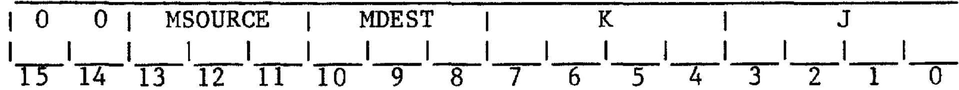
```
 _________________________________________________________________
|                                       |Adv|Dir|  pt   |       J       |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
 15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
```

pt is the port number; specifies one of 4 ports
Dir=0 for output port; Dir=1 for input port
Adv=1 to advance port (remove word from fifo) after input port is read

```
KEY:  Rn    is register number n.
      Rn++  is register number n, incremented after reading.
      --Rn  is register number n, decremented before reading.
      val   is an immediate value.
      @z    is the memory word whose address is z.
      A | B    means concatenation of bit field  A  and bit field  B .
      f<i>     means the i-th bit of  f .  f<i:j> means i-th to j-th bits of  f .
               Bits are numbered from least to most significant.
      FPC  is the flag/PC word:  C | V | N | Z | PC
```

```
SPECIAL CASES:   HARD RESET:    (Reset pin goes HIGH)
                                0 -> PC
                 SOFT RESET:    (Interrupt pin goes HIGH for >=26 microcycles)
                                FPC -> @(-1);   0 -> PC
                 INTERRUPT:     (Interrupt pin goes HIGH for 1 microcycle)
                                FPC -> @(-2);   @(-3) -> PC
```

MOVE INSTRUCTIONS:



Assembly syntax: MOVE <source>,<destination>
Execution time = 3 + Time(MSOURCE) + Time(MDEST) microcycles
Ri means Rk when MSOURCE is 0, 1, 2, or 3; Ri means Rj otherwise.

All move instructions modify Z and N based on value of X, and set V to zero.

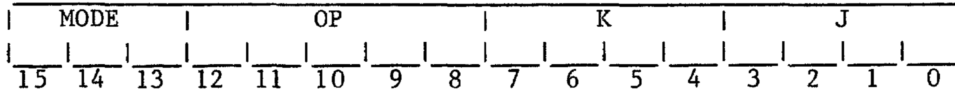
MSOURCE	X	<source>		Time(MSOURCE)
0	Rj	Rj		0
1	@Rj	@Rj		2
2	@Rj++	@Rj++		2
3	@(Rj+val)	@Rj+val		3
4	val	#val		0
5	@val	@#val		2
6	Input Port pt	Ppt+	[advance]	1
		Ppt=	[no advance]	1
7	0	0		0

MDEST	effect	<destination>	Time(MDEST)
0	X -> Ri	Ri	0
1	X -> @Ri	@Ri	2
2	X -> @Ri++	@Ri++	2
3	X -> @(Ri+val)	@Ri+val	4
4	X -> @--Ri	@--Ri	3
5	X -> @val	@#val	3
6	X -> Output Port pt	Ppt	0
7	X -> nowhere	.	0

Notes: When source and destination both contain a val, then they are different (i.e. the instruction takes two immediate values).
When source or destination is a port, instruction does not terminate until port is ready (i.e. until input port has a word to read or output port has room for another word).
When source is a port, destination may not be a port due to contention for field k.

MOSAIC PROCESSOR VERSION B INSTRUCTION SET

ARITHMETIC AND BRANCH INSTRUCTION MODES:



Execution time = 3 + Time(MODE) + Time(OP)

MODE	X	Y	Dest	Assembly language syntax (when Rk not specified, k=j)	Time(MODE)
2	Rj	Rk	Rk	<OP Mnemonic> Rj {, Rk}	0
3	val	Rj	Rk	<OP Mnemonic> #val , Rj {, Rk}	0
4	@Rj	Rk	Rk	<OP Mnemonic> @Rj {, Rk}	2
5	@val	Rj	Rk	<OP Mnemonic> @#val , Rj {, Rk}	2
6	@Rj	Rk	@Rj	<OP Mnemonic> M @Rj {, Rk}	4
7	@val	Rj	@val	<OP Mnemonic> M @#val {, Rj}	4

Modes 6 and 7 are defined only for OPs that assign a result to Dest.

BRANCH CONDITIONS:

k	Condition		Alternate <OP Mnemonic> (implies OP=BRAT)	(implies OP=BRAF)
00pt	Output port number pt Not Ready (i.e. no room in port to do output)		BONR	BOR
01pt	Input port number pt Not Ready (i.e. no word in input port to read)		BINR	BIR
1000	V	[overflow]	BVS	BVC
1001	N	[negative]	BNS	BNC
1010	~C	[Carry = 0]	BCC or BLO	BCS or BHIS
1011	N xor V	[signed <]	BLT	BGE
1100	Z	[zero]	BZS or BEQ	BZC or BNE
1101	Z or N	[<= zero]	BLEZ	BGTZ
1110	Z or ~C	[unsigned <=]	BLOS	BHI
1111	Z or (N xor V)	[signed <=]	BLE	BGT

ARITHMETIC INSTRUCTIONS:

All Arithmetic instructions modify the Z, N, and V flags.
Some always set V to 0. They are:
ASR, ROR, LSR, RNR, AND, OR, XOR, COM, BITT, and MUL

OP	Instruction	<OP Mnemonic>	Effect	C flag modified?	TIME(OP)
0	INCrement	INC	X + 1 -> Dest	no	0
1	DECrement	DEC	X - 1 -> Dest	no	0
2	Arithmetic Shift Right	ASR	X<15>|X -> Dest|C	yes	0
3	Arithmetic Shift Left	ASL	X + X -> Dest	yes	0
4	ROtate Right	ROR	C|X -> Dest|C	yes	0
5	ROtate Left	ROL	X + X + C -> Dest	yes	0
6	Logical Shift Right	LSR	0|X -> Dest|C	yes	0
7	Rotate Nibble Right	RNR	X<3:0>|X<15:4> -> Dest	no	0
8	ADD	ADD	X + Y -> Dest	yes	0
9	ADD with Carry	ADDC	X + Y + C -> Dest	yes	0
A	SUBtract	SUB	Y - X -> Dest	yes	0
B	SUBtract with Carry	SUBC	Y - X - 1 + C -> Dest	yes	0
C	SUBtract Negate	SUBN	X - Y -> Dest	yes	0
D	SUBtract Negate with Carry	SUBNC	X - Y - 1 + C -> Dest	yes	0
E	NEGate	NEG	-X -> Dest	yes	0
F	INCrement with Carry	INCC	X + C -> Dest	yes	0
10	bitwise COMplement	COM	~X -> Dest	no	0
11	bitwise AND	AND	X and Y -> Dest	no	0
12	bitwise OR	OR	X or Y -> Dest	no	0
13	bitwise eXclusive OR	XOR	X exor Y -> Dest	no	0
14	CoMPare	CMP	X - Y	yes	0
15	BIT Test	BITT	X and Y	no	0
16	unsigned MULtiply	MUL	high word(X*Y) -> RJ low word(X*Y) -> RK modify Z, N, V based on high word	no	18
17	undefined				

BRANCH INSTRUCTIONS:

OP	Instruction	<OP Mnemonic>	Effect	TIME(OP)
18	JUMP	JUMP	X -> PC	1
19	Jump and ReSTore flags	JRST	X -> FPC	1
1A	POP Jump	POPJ	@Rk++ -> PC	3
1B	POP Jump and Restore flags	POPJR	@Rk++ -> FPC	3
1C	Branch True	BRAT	X -> PC if Condition k true	0 or 1 *
1D	Branch False	BRAF	X -> PC if Condition k false	0 or 1 *
1E	PUSH Jump	PUSHJ	Y-1 -> Rk; FPC -> @Rk; X -> PC	4
1F	undefined			

* TIME(OP) = 0 if branch is not taken; 1 if branch is taken.

When <OP Mnemonic> is BONR, BOR, BINR, or BIR:
pt may be specified by writing "Ppt" in place of "Rk" field.

```
! Mosaic ver. B microcode     page 1 of 7

  Version 15-MAR-84
! Syntax of implicants is:
! word  <inputs>  ::  <outputs>

! MACROS:
DEF  Cin=0        Cforce
DEF  Cin=1        Cforce Cvall
DEF  PC++->A      PC->inc Addl inc->A A->PC
DEF  RA++->A      RA->inc Addl inc->A A->RA
DEF  PC->A        PC->inc inc->A A->PC
DEF  RJ=>         useJ R=>
DEF  RK=>         R=>
DEF  RJ           useJ R
DEF  RK           R
DEF  X=>          GP= 0C Cin=0 nosh W=>
DEF  Y=>          GP= 0A Cin=0 nosh W=>
DEF  X+Y=>        GP= 86 Cin=0 nosh W=>
DEF  Y+1=>        GP= 0A Cin=1 nosh W=>
DEF  Y-1=>        GP= A5 Cin=0 nosh W=>
DEF  X+1=>        GP= 0C Cin=1 nosh W=>
DEF  X-1=>        GP= C3 Cin=0 nosh W=>
DEF  -1=>         GP= 0F Cin=0 nosh W=>
DEF  0=>          GP= 00 Cin=0 nosh W=>
    ! rnib  refreshes the carry flag.  This is done on the cycle
    ! after DISPATCH: of every instruction.
DEF  saveC        rnib
DEF  testX        GP= 0C Cin=0 nosh setZNV

! FEEDBACK MNEMONICS:
DEF  .reset2         FB= 0 0 0 0 0
DEF  .reset3         FB= 0 0 0 0 1
DEF  .reset4         FB= 0 0 0 1 0
DEF  .interrupt2     FB= 0 0 0 1 1
DEF  .interrupt3     FB= 0 0 1 0 0
DEF  .interrupt4     FB= 0 0 1 0 1
DEF  .interrupt5     FB= 0 0 1 1 0
DEF  .interrupt6     FB= 0 0 1 1 1
DEF  .refetch        FB= 0 1 0 0 0
DEF  .fetch          FB= 0 1 0 0 1
DEF  .decode         FB= 0 1 0 1 0
DEF  .get            FB= 0 1 1 0 0
DEF  .get2           FB= 0 1 1 0 1
DEF  .get3           FB= 0 1 1 1 0
DEF  .get4           FB= 0 1 1 1 1
DEF  .go             FB= 1 0 0 0 0
DEF  .go2            FB= 1 0 0 0 1
DEF  .go3            FB= 1 0 0 1 0
DEF  .mov            FB= 1 0 0 1 1
DEF  .mov2           FB= 1 0 1 0 0
DEF  .mov3           FB= 1 0 1 0 1
DEF  .store          FB= 1 0 1 1 0
DEF  .RJ++--         FB= 1 0 1 1 1
DEF  .PC-2           FB= 1 1 0 0 0
```

```
! Mosaic ver. B microcode     page 2 of 7


!     CONTROLLER INPUTS (21):

! 'FB= <fb4> <fb3> ... <fb0>'   5 Feedback bits
! 'I=  <i15> <i14> ... <i7>'   10 Instruction register bits
!                                 Unspecified bits are '*' (don't care).
!                                 When 'I=' not specified for an implicant,
!                                   that from last implicant is used.
! 'RESET'                         Hard reset  ('RESET=0' implied when
!                                   'RESET=1' not specified)
! 'INT'                           External interrupt flip flop set
! 'FlagC'                         Flag Condition
! 'PortC'                         Port Condition (0 when port K is ready)
! 'Mout'                          Shift out of Multiplier/Product register
! 'SRout'                         Shift out of 16-bit shift register

   ! CONTROLLER OUTPUTS (49):

! 'FB= <fb4> <fb3> ... <fb0>'   5 Feedback bits
   ! Bus sources:
! 'IN=>'  Memory data input           'M=>'  Multiplier/product
! 'PC=>'  Program Counter             'W=>'  ALU/shifter output
! 'Pt=>'  Input port pt
! 'R=>'   If useJ then Register J, else Register K
   ! Bus destinations:
! 'D'   Memory Data out               'A'  Memory Address
! 'F'   Flags (C,V,N,Z)               'M'  Multiplier/Product
! 'X'   ALU X input                   'Y'  ALU Y input
! 'Yshift'  ALU Y input, bus data shifted right, Carry from ALU into high bit
! 'R'   If useJ then Register J, else Register K
! 'Pt'  Output port pt
   ! ALU/Shifter control:
! 'GP= <hex digit><hex digit>'  Carry Generate and Propogate codes (7 bits)
!         Bits of each digit are for: (X=1,Y=1)(X=1,Y=0)(X=0,Y=1)(X=0,Y=0).
!         LSB of G code (X=0,Y=0) must be zero.
! 'Cforce' Carry in to ALU is (IF Cforce then (IF Cvall then 1 else 0)
! 'Cvall'                                  else old Carry flag)
! 'asr'   Arithmetic shift right    'lsr'  Logical shift right
! 'ror'   Rotate right              'rnib' Rotate nibble
! 'nosh'  No shift, and recirculate Carry flag
! 'setC'  Set C flag to Carry out   'setZNV' Modify Z, N, and V flags
! 'Mshift' Shift Multiplier/Product right
   ! Address section
! 'PC->inc' PC goes to incrementer   'RA->inc'  Refresh address to incrementer
! 'Addl'    carry in to incrementer is 1 (defaults to 0)
! 'inc->A'  incrementer output goes to memory address
! 'A->PC'   Address goes to PC       'A->RA'    Address goes to RA
   ! Miscellaneous
! 'INT:=0' Clear external interrupt flip flop
! 'IN->I'  Latch instruction register with new memory data
! 'WRITE'  Write data in D to memory location in A
! 'SRin=1' Inject bit into shift register for counting multiply steps
! 'useJ'   Let field J (as opposed to K) select a register
! 'Advance' Advance input port pt
```

```
! HARD RESET
   ! X and Y are assigned to make them digital, so ALU can make constants later
word hardreset:  RESET=1  FB= *  I= *      ::  INT:=0  X Y  RA++->A  .reset4

! INTERRUPT AND SOFT RESET

  ! Interrupt:  PCF-1 -> @(-1);  @(-2) -> PC.
word interrupt1: .decode     I= * INT=1  ::  INT:=0   PC=> X       .interrupt2
word interrupt2: .interrupt2             ::           X-1=> D      .interrupt3
word interrupt3: .interrupt3             ::            -1=> A X    .interrupt4
word interrupt4: .interrupt4      INT=0  ::  Write    X-1=> A      .interrupt5
word interrupt5: .interrupt5             ::                        .interrupt6
word interrupt6: .interrupt6             ::    IN=> A A->PC        .fetch

  ! If interrupt persists, do a soft reset:  PCF-1 -> @(-3);  0 -> PC.
word softreset:  .interrupt4      INT=1  ::           X-1=> X      .reset2
word reset2:     .reset2                 ::           X-1=> A      .reset3
word reset3:     .reset3                 ::  Write                 .reset4
  ! While waiting for INT=0, keep storage static and X and Y digital:
word reset4:     .reset4          INT=1  ::  INT:=0  X Y  RA++->A  .reset4
word reset4:     .reset4          INT=0  ::     0=> A A->PC        .fetch


! FETCH AND DECODE

word fetch:    .fetch  I= *       ::                     PC++->A  .decode
       ! All instructions pass through  decode:  (or  interrupt1: )
word decode:   .decode I= * INT=0 :: IN->I  saveC  RJ=> X Y D M  RA++->A .get
```

```
! Mosaic ver. B microcode     page 4 of 7


! REFETCH ON FAILED I/O IN MOVES

word refetch:  .refetch   I= *          :: X-1=> A  A->PC           .fetch


! MOVE SOURCES USING REGISTER J (SO DESTINATION USES K)

word RJ->:       .get       I= 0 0 0 0 0  ::                   PC->A    .mov

word @RJ->:      .get       I= 0 0 0 0 1  ::           RJ=> A            .get3

word @RJ++->:    .get       I= 0 0 0 1 0  ::           RJ=> A Y          .get2
word @RJ++->2:   .get2                    ::          Y+1=> RJ   PC->A    .get4

word @(RJ+#)->:  .get       I= 0 0 0 1 1  ::           IN=> Y    PC++->A .get2
word @(RJ+#)->2: .get2                    ::          X+Y=> A            .get3

word @-wait:   .get3        I= 0 0 * * *  ::                     PC->A    .get4
word any-@:    .get4        I= 0 0 * * *  ::           IN=> X D          .mov


! MOVE SOURCES NOT USING REGISTER J (SO DESTINATION USES J)

word #->:       .get        I= 0 0 1 0 0  ::           IN=> X D  PC++->A .mov

word @#->:      .get        I= 0 0 1 0 1  ::           IN=> A            .get2
word @#->2:     .get2                     ::                   PC++->A   .get4

word badPt->:  .get           I= 0 0 1 1 0 * * * * * 0 ::        PC=> X .refetch
    ! Wait a cycle so controller can dispatch on port condition:
word InPt->:    .get          I= 0 0 1 1 0 * * * * * 1 ::            PC->A .get2
word InPt->:    .get2 PortC=0 I= 0 0 1 1 0 * * * * 0 1 :: Pt=> X D          .mov
word InPtA->:   .get2 PortC=0 I= 0 0 1 1 0 * * * * 1 1 :: Pt=> X D Advance .mov
word iofailed: .get2 PortC=1 I= 0 0 1 1 0              :: PC=> X         .refetch

word 0->:      .get        I= 0 0 1 1 1  ::            0=> X D  PC->A    .mov
```

```
! Mosaic ver. B microcode     page 5 of 7


! STUFF ASSOCIATED WITH MOVE DESTINATIONS

   ! Use register J in next two cycles only if MOVE source didn't use it:
word pickJ:     .mov    I= 0 0 1 * * * * *  :: NOTRANSISTORS  useJ
word pickJ2:    .mov2   I= 0 0 1 * * * * *  :: NOTRANSISTORS  useJ

word store:     .store  I= 0 0              :: WRITE      PC->A        .fetch


! MOVE DESTINATIONS PROPER

word ->R:       .mov    I= 0 0 * * * 0 0 0 ::  X=> R  setZNV  PC++->A  .decode
word ->@R:      .mov    I= 0 0 * * * 0 0 1 ::  testX  R=> A            .store

word ->@R++:    .mov    I= 0 0 * * * 0 1 0 ::  testX  R=> A X          .mov2
word ->@R++2:   .mov2                      ::  Write  X+1=> R  PC->A   .fetch

word ->@(R+#):  .mov    I= 0 0 * * * 0 1 1 ::  testX     R=> X         .mov
word ->@(R+#)2: .mov2                      ::   IN=> Y    PC++->A      .mov3
word ->@(R+#)3: .mov3                      ::  X+Y=> A                 .store

word ->@--R:    .mov    I= 0 0 * * * 1 0 0 ::  testX     R=> X         .mov2
word ->@--R2:   .mov2                      ::  X-1=> A R               .store

    ! Note: Wait a cycle before taking #value from IN to guarantee correct data
    ! independent of MOVE source:
word ->@#:      .mov    I= 0 0 * * * 1 0 1 ::  testX         PC++->A   .mov2
word ->@#2:     .mov2                      ::  IN=> A                  .store

   ! MOVE to output port: if K specifies input port, then do nothing;
   !                       if port isn't ready, reverse side effects and refetch
word ->Pt:  PortC=0 .mov I= 0 0 * * * 1 1 0 * 0 :: X=> Pt setZNV PC++->A .decode
word ->badPt:       .mov I= 0 0 * * * 1 1 0 * 1 ::        setZNV PC++->A .decode
word ->Ptwait: PortC=1 .mov I= 0 0 0 0 * 1 1 0  ::  PC=> X           .refetch
word ->Ptwait: PortC=1 .mov I= 0 0 0 1 0 1 1 0   ::   Y=> RJ         .RJ++--
word RJ++--:  .RJ++--                            ::  PC=> X           .refetch
word ->Ptwait: PortC=1 .mov I= 0 0 0 1 1 1 1 0   ::  PC=> X           .PC-2
word ->Ptwait: PortC=1 .mov I= 0 0 1 0 * 1 1 0   ::  PC=> X           .PC-2
word PC-2:    .PC-2         I= *                 :: X-1=> X           .refetch
word ->Ptwait: PortC=1 .mov I= 0 0 1 1 * 1 1 0   ::  PC=> X           .refetch

word ->*:       .mov        I= 0 0 * * * 1 1 1   ::  testX  PC++->A   .decode
```

```
! Mosaic ver. B microcode     page 6 of 7


! ARITHMETIC AND BRANCH SOURCES

word J,K,K:     .get  I= 0 1 0      ::          PC->A    RK=> Y       .go
word #,J,K:     .get  I= 0 1 1      ::          PC++->A  IN=> X       .go

word @J,K,:     .get  I= 1 * 0      ::                   RJ=> A       .get2
word @J,K,K:    .get2 I= 1 0 0      ::          PC->A    RK=> Y       .get3
word @J,K,@J:   .get2 I= 1 1 0      ::                   RK=> Y       .get3

word @#,J,:     .get  I= 1 * 1      ::                   IN=> A       .get2
word @#,J,K:    .get2 I= 1 0 1      ::          PC++->A               .get3
word @#,J,@#:   .get2 I= 1 1 1      ::                                .get3

word any@:      .get3 I= 1 * *      ::                   IN=> X       .go


! WORDS TO SPECIFY ALU FUNCTION IN NORMAL ARITHMETICS

word inc:    .go  I= * * * 0 0 0 0 0  ::  ALUONLY GP= 0C Cin=1 nosh setZNV
word dec:    .go  I= * * * 0 0 0 0 1  ::  ALUONLY GP= C3 Cin=0 nosh setZNV
word asr:    .go  I= * * * 0 0 0 1 0  ::  ALUONLY GP= 0C Cin=0 asr  setZNV
word asl:    .go  I= * * * 0 0 0 1 1  ::  ALUONLY GP= C0 Cin=0 nosh setZNV setC
word ror:    .go  I= * * * 0 0 1 0 0  ::  ALUONLY GP= 0C Cin=0 ror  setZNV
word rol:    .go  I= * * * 0 0 1 0 1  ::  ALUONLY GP= C0       nosh setZNV setC
word lsr:    .go  I= * * * 0 0 1 1 0  ::  ALUONLY GP= 0C Cin=0 lsr  setZNV
word rnr:    .go  I= * * * 0 0 1 1 1  ::  ALUONLY GP= 0C Cin=0 rnib setZNV
word add:    .go  I= * * * 0 1 0 0 0  ::  ALUONLY GP= 86 Cin=0 nosh setZNV setC
word addc:   .go  I= * * * 0 1 0 0 1  ::  ALUONLY GP= 86       nosh setZNV setC
word sub:    .go  I= * * * 0 1 0 1 0  ::  ALUONLY GP= 29 Cin=1 nosh setZNV setC
word subc:   .go  I= * * * 0 1 0 1 1  ::  ALUONLY GP= 29       nosh setZNV setC
word subn:   .go  I= * * * 0 1 1 0 0  ::  ALUONLY GP= 49 Cin=1 nosh setZNV setC
word subnc:  .go  I= * * * 0 1 1 0 1  ::  ALUONLY GP= 49       nosh setZNV setC
word neg:    .go  I= * * * 0 1 1 1 0  ::  ALUONLY GP= 03 Cin=1 nosh setZNV setC
word incc:   .go  I= * * * 0 1 1 1 1  ::  ALUONLY GP= 0C       nosh setZNV setC
word com:    .go  I= * * * 1 0 0 0 0  ::  ALUONLY GP= 03 Cin=0 nosh setZNV
word and:    .go  I= * * * 1 0 0 0 1  ::  ALUONLY GP= 08 Cin=0 nosh setZNV
word or:     .go  I= * * * 1 0 0 1 0  ::  ALUONLY GP= 0E Cin=0 nosh setZNV
word xor:    .go  I= * * * 1 0 0 1 1  ::  ALUONLY GP= 06 Cin=0 nosh setZNV


! NORMAL ARITHMETIC DESTINATIONS

word ALU->K:    .go     I= 0 1 * 0 * * * * ::  PC++->A  NOALU  W=> RK   .decode
word ALU->K:    .go     I= 0 1 * 1 0 0 * * ::  PC++->A  NOALU  W=> RK   .decode
word ALU->K:    .go     I= 1 0 * 0 * * * * ::  PC++->A  NOALU  W=> RK   .decode
word ALU->K:    .go     I= 1 0 * 1 0 0 * * ::  PC++->A  NOALU  W=> RK   .decode

word ALU->@:    .go     I= 1 1 * 0 * * * * ::           NOALU  W=> D    .store
word ALU->@:    .go     I= 1 1 * 1 0 0 * * ::           NOALU  W=> D    .store
word ALU->@J:   .store  I= 1 1 0           ::  Write  PC->A                .fetch
word ALU->@#:   .store  I= 1 1 1           ::  Write  PC++->A              .fetch
```

```
! Mosaic ver. B microcode     page 7 of 7


! SPECIAL ARITHMETICS:  MULTIPLY, ETC.

word cmp:  .go  I= * * * 1 0 1 0 0      :: PC++->A GP= 49 Cin=1 nosh setZNV setC
                                                                     .decode
word bitt: .go  I= * * * 1 0 1 0 1      :: PC++->A GP= 08 Cin=0 nosh setZNV
                                                                     .decode
word mul:  .go  I= * * * 1 0 1 1 *      :: Mshift  0=> Y  SRin=1              .go2
word mul0:     .go2  SRout=0  Mout=0    :: Mshift    Y=> Yshift  RA++->A  .go2
word mul1:     .go2  SRout=0  Mout=1    :: Mshift  X+Y=> Yshift  RA++->A  .go2
word muldone: .go2  SRout=1             :: Mshift    Y=> RJ setZNV PC->A  .go3
word mulend:   .go3                     ::           M=> RK       PC++->A  .decode


! BRANCHES

word jump:        .go    I= * * * 1 1 0 0 0  ::  X=>   A A->PC            .fetch
word jrst:        .go    I= * * * 1 1 0 0 1  ::  X=> F A A->PC            .fetch

word popj:        .go    I= * * * 1 1 0 1 *  ::  RK=> Y A                 .go2
word popj2:       .go2                       ::  Y+1=> RK                 .go3
word popj3:       .go3   I= * * * 1 1 0 1 0  ::  IN=>   A A->PC           .fetch
word popjr:       .go3   I= * * * 1 1 0 1 1  ::  IN=> F A A->PC           .fetch

word /->PC(P=0): .go  PortC=0  I= * * * 1 1 1 0 0 0  ::  PC++->A       .decode
word ->PC(P=1):  .go  PortC=1  I= * * * 1 1 1 0 0 0  ::  X=> A  A->PC  .fetch
word /->PC(F=0): .go  FlagC=0  I= * * * 1 1 1 0 0 1  ::  PC++->A       .decode
word ->PC(F=1):  .go  FlagC=1  I= * * * 1 1 1 0 0 1  ::  X=> A  A->PC  .fetch
word /->PC(P=1): .go  PortC=1  I= * * * 1 1 1 0 1 0  ::  PC++->A       .decode
word ->PC(P=0):  .go  PortC=0  I= * * * 1 1 1 0 1 0  ::  X=> A  A->PC  .fetch
word /->PC(F=1): .go  FlagC=1  I= * * * 1 1 1 0 1 1  ::  PC++->A       .decode
word ->PC(F=0):  .go  FlagC=0  I= * * * 1 1 1 0 1 1  ::  X=> A  A->PC  .fetch

word ->PushJ:     .go    I= * * * 1 1 1 1 *  ::  Y-1=> RK A               .go2
word ->PushJ2:    .go2                       ::   PC=> D                  .go3
word ->PushJ3:    .go3                       ::  Write  X=> A A->PC       .fetch
```

MOSAIC VERSION B CIRCUIT DIAGRAMS 4-84

Same as Mosaic ver. A except:

1. As indicated in ver. A Instruction register and flags
2. Replacing controller
3. Including SR and M registers; replacing Y register
4. Including Interrupt flip flop

Additional Pads:

φ_{1L} φ_{2L} INT Pad

Additional Controller Inputs:

INTff SRout Mout FB<4>

Additional Controller Outputs:

* indicates type "raw". All others type "C1"

M ⇒

⇒ M ⇒ Yshift ⇒ F

Mshift*

SRin*

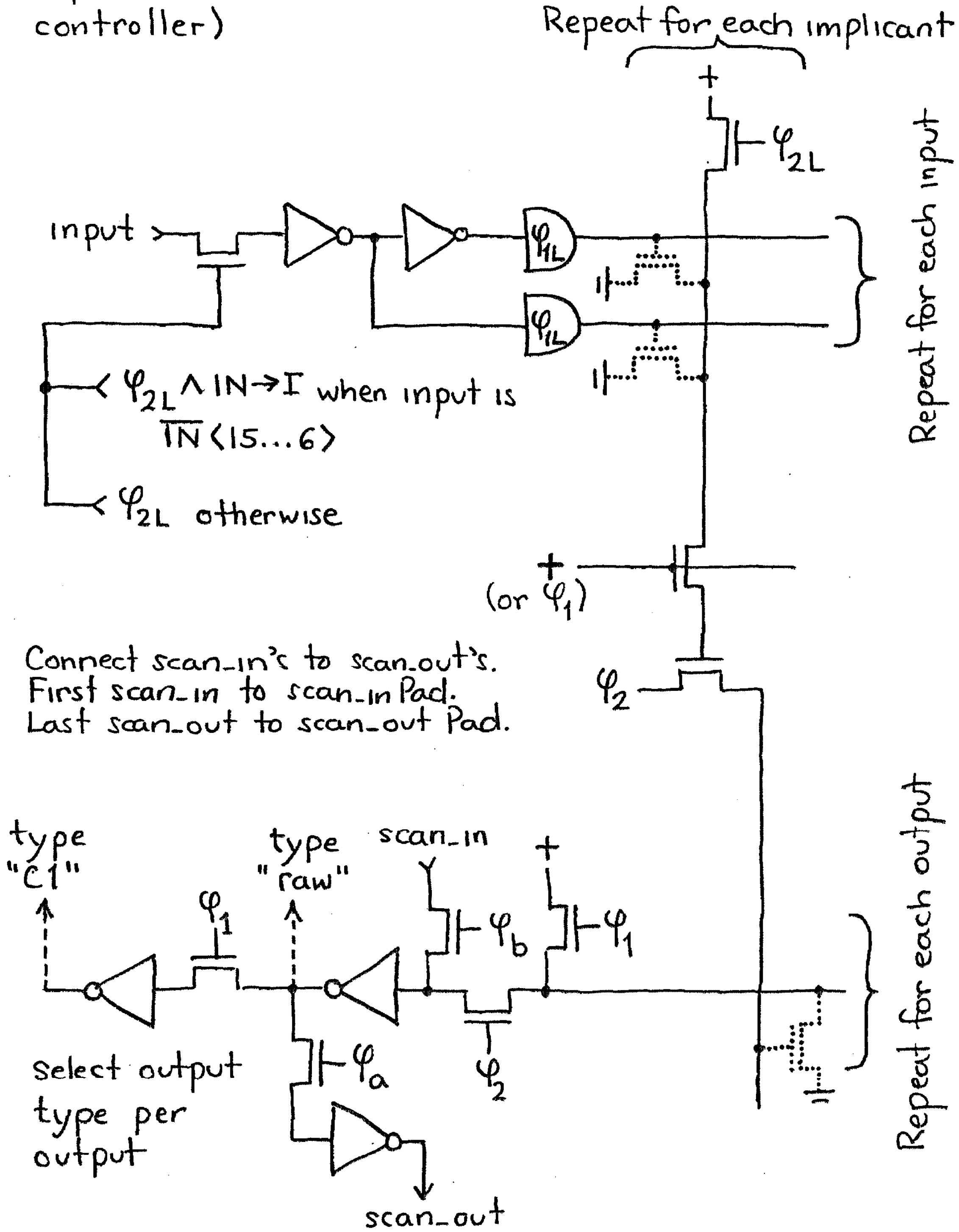
INT ← Ø

FB<4>

Controller

Mosaic ver. B 4-84

(Replaces Mosaic ver. A controller)

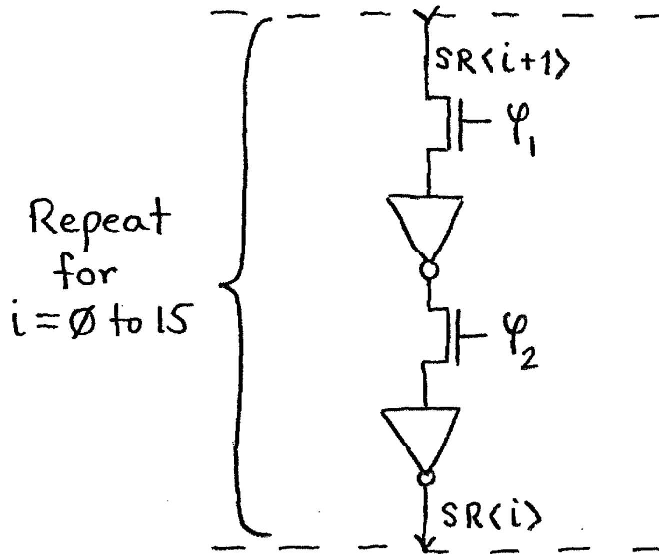


SR AND M REGISTERS

Mosaic ver. B 4-84

SR Shift Register for counting multiply steps

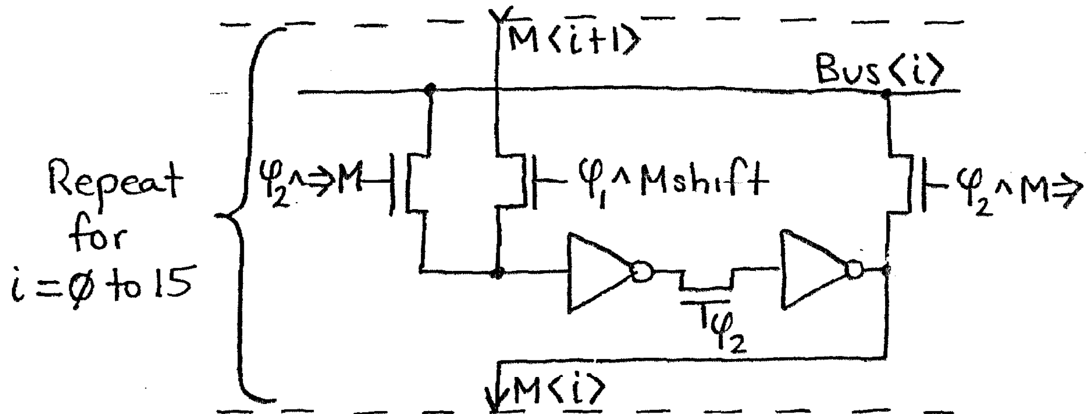
SR<16> = SRin



SR<∅> = SRout

M Multiplier/Product Register

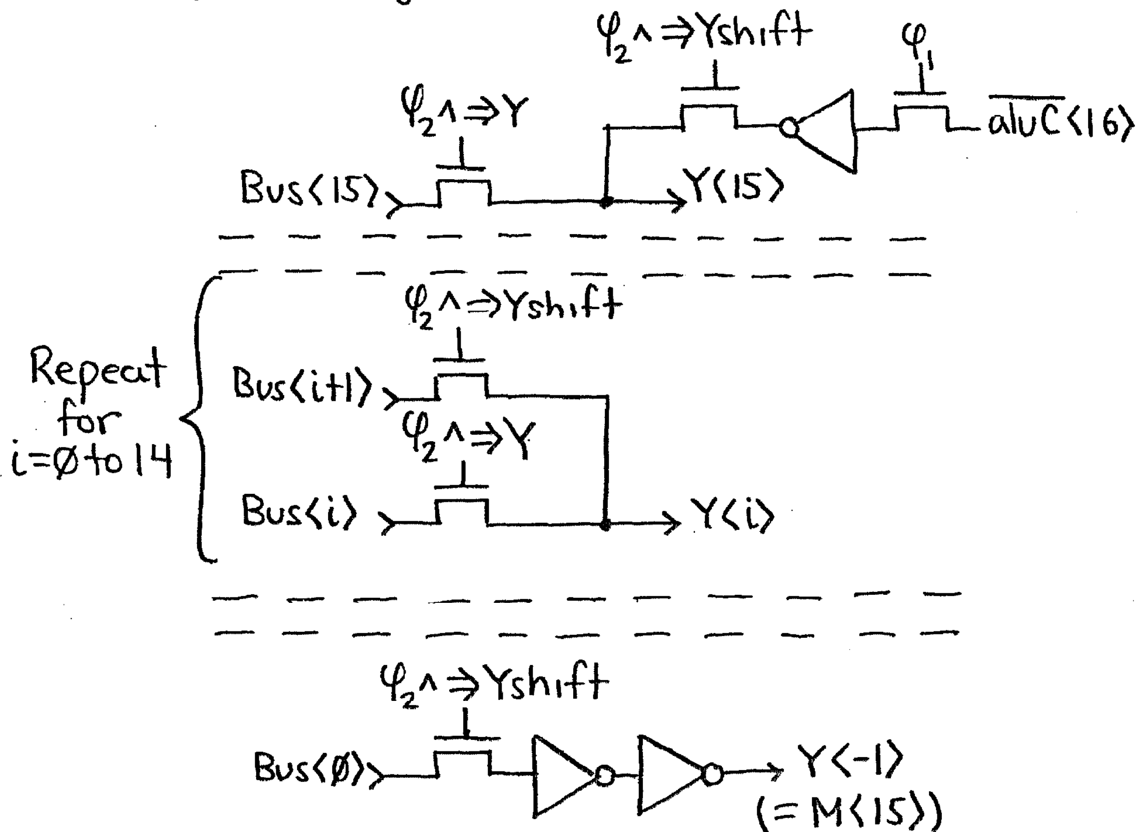
M<16> = Y<-1>



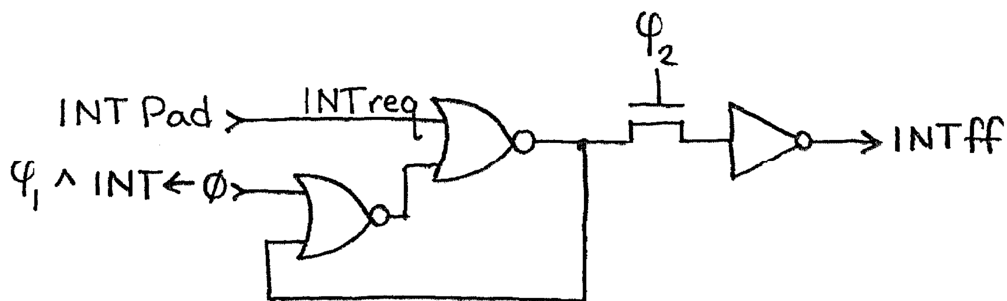
M<∅> = Mout

Y register (and Interrupt Flip Flop) Mosaic ver. B 4-84

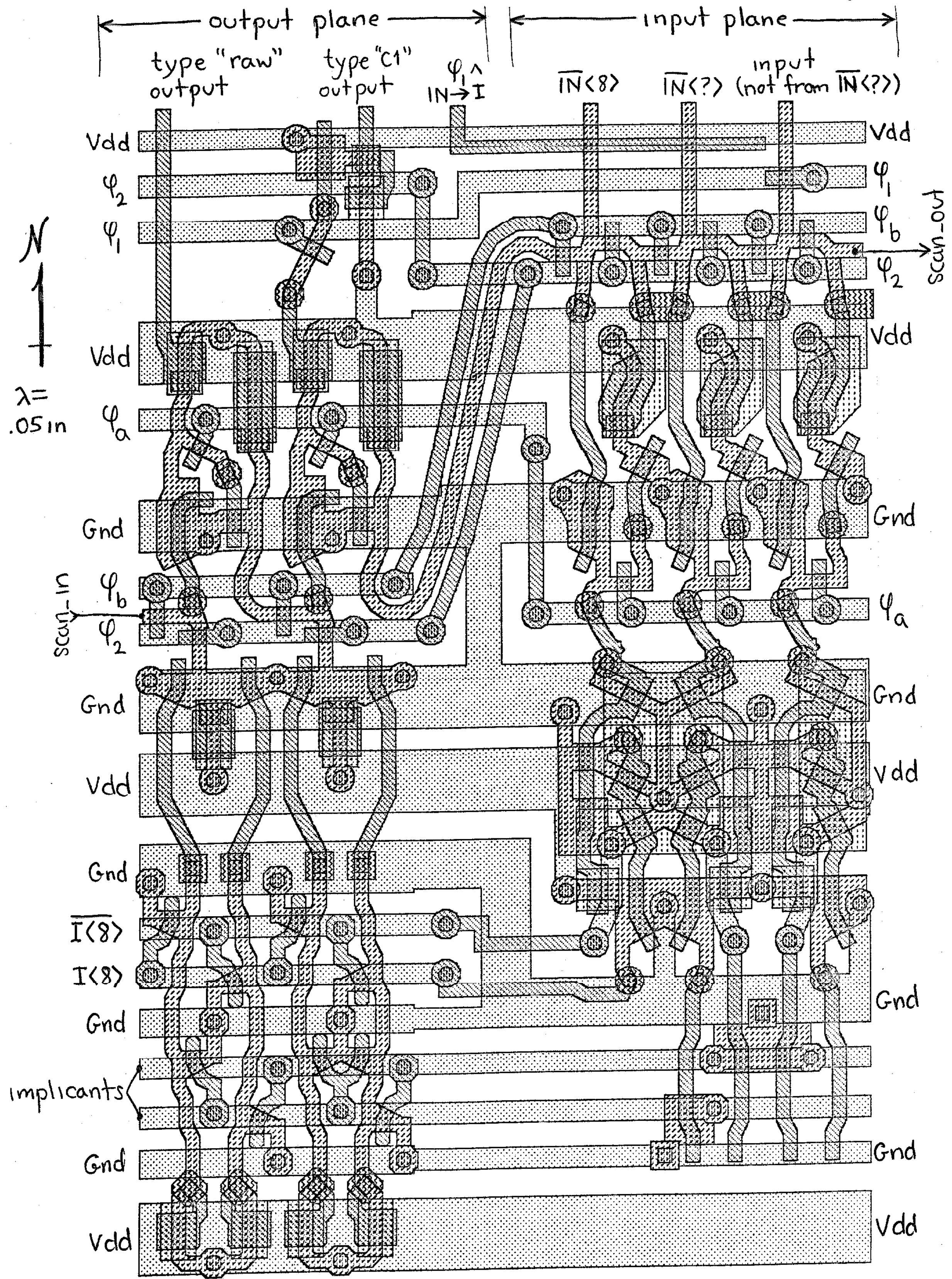
[Y] (Replaces Y register of version A)



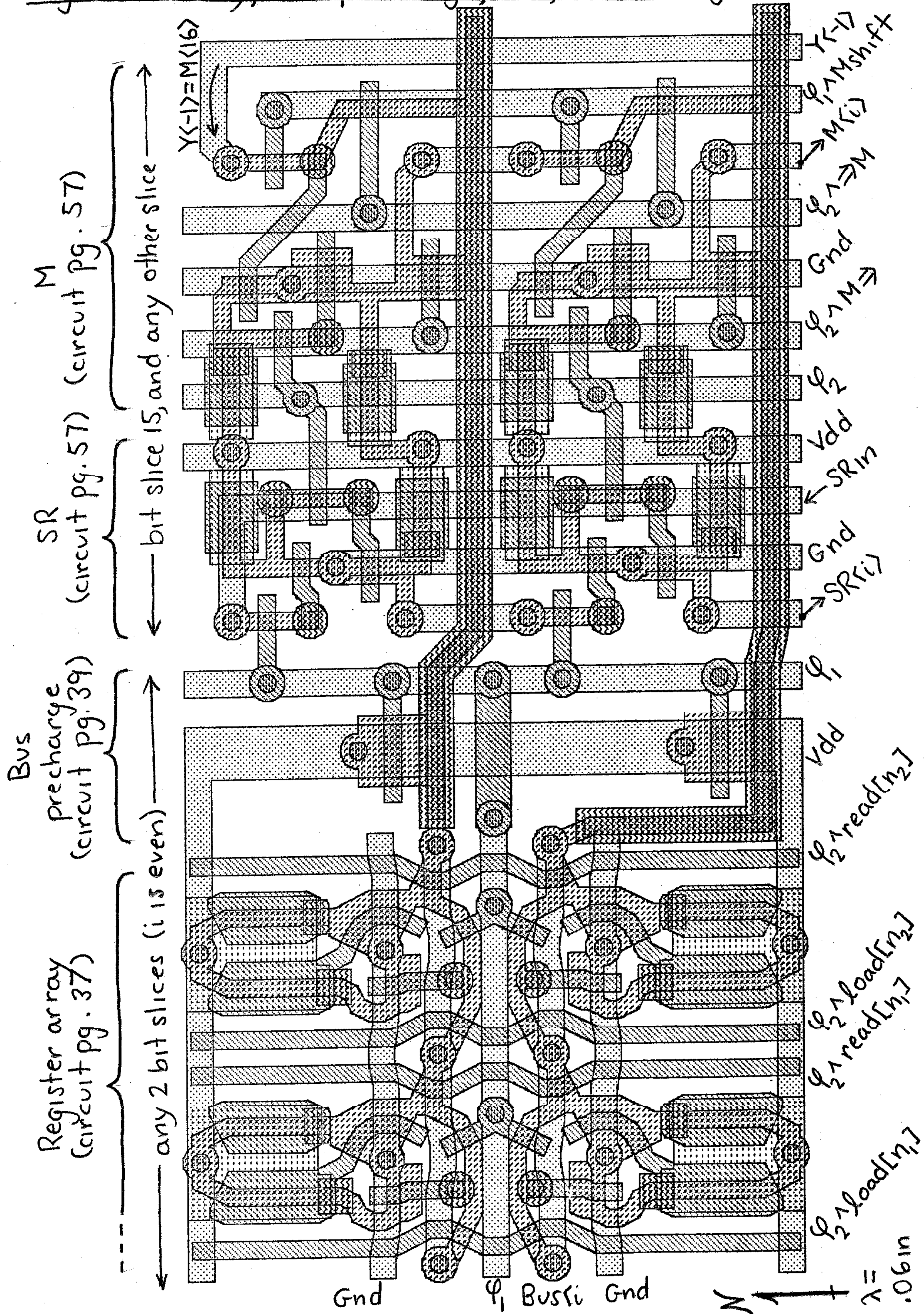
Interrupt flip flop



APPENDIX C: SELECTED LAYOUT Controller (circuit pg. 30)



Register array, Bus precharge, SR, and M registers



ALU input latches and function blocks (circuit pg. 31)

ALU carry chain

Propagate block

Generate block

any 2 bit slices

Input latches (see modification to Y, pg 58)

M register

λ $\lambda = .06$ in

G<i+1> P<i> Bus<i+1> G<i> P<i> Bus<i>

$\varphi_1 \wedge P00$ (XY)

$\varphi_1 \wedge P01$

$\varphi_1 \wedge P11$

$\varphi_1 \wedge P10$

$\varphi_1 \wedge G01$

$\varphi_1 \wedge G11$

$\varphi_1 \wedge G10$

G00 (=Gnd)

Gnd

Vdd

Gnd

$\varphi_2 \wedge \Rightarrow X$

$\varphi_2 \wedge \Rightarrow Y$

$\varphi_2 \wedge \Rightarrow Y$shift

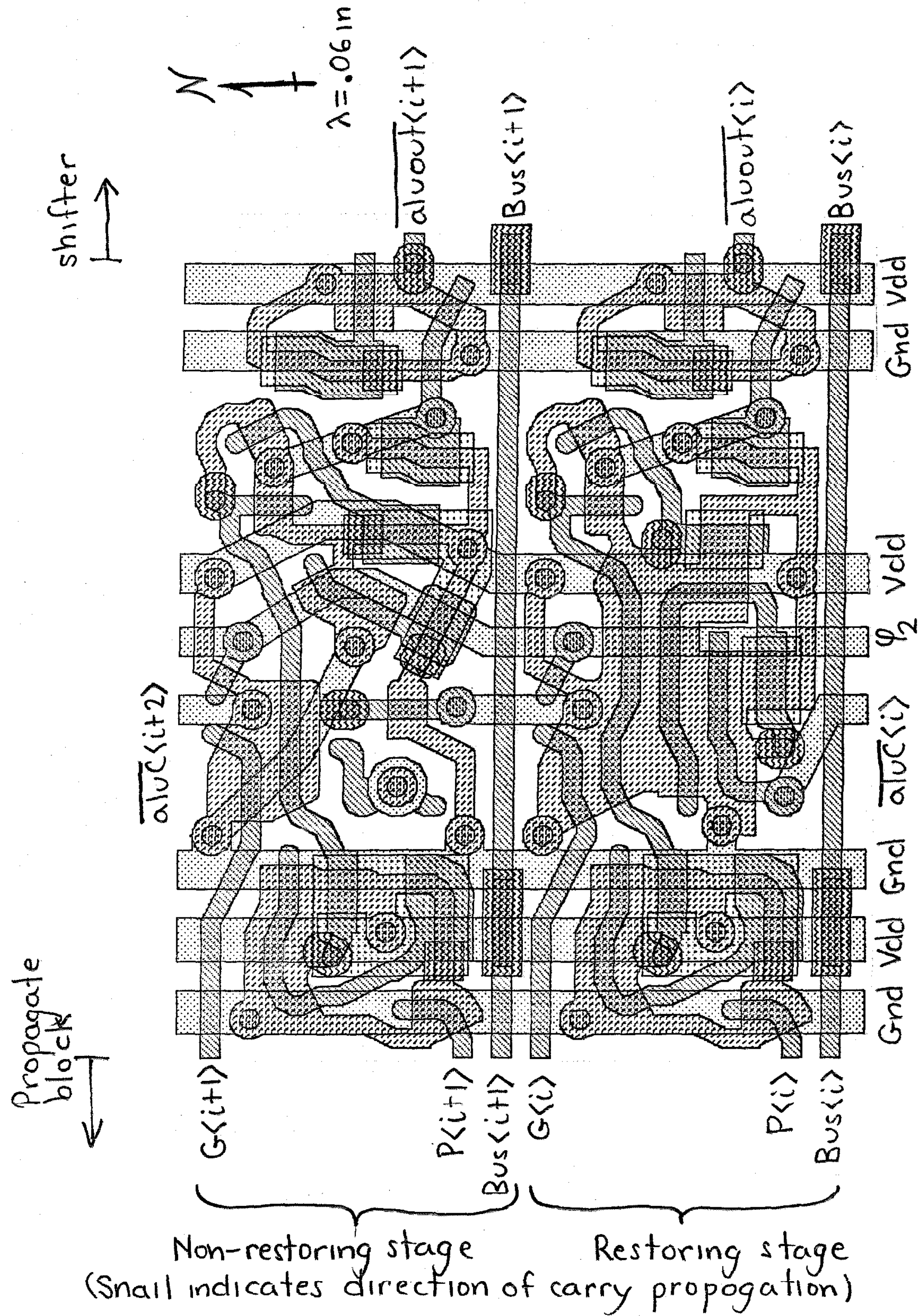
Bus<i+2> (unless i=15)

Y̅ Y X̅ X Y Y̅ X̅ X

Y X

Bus<i+1> Bus<i>

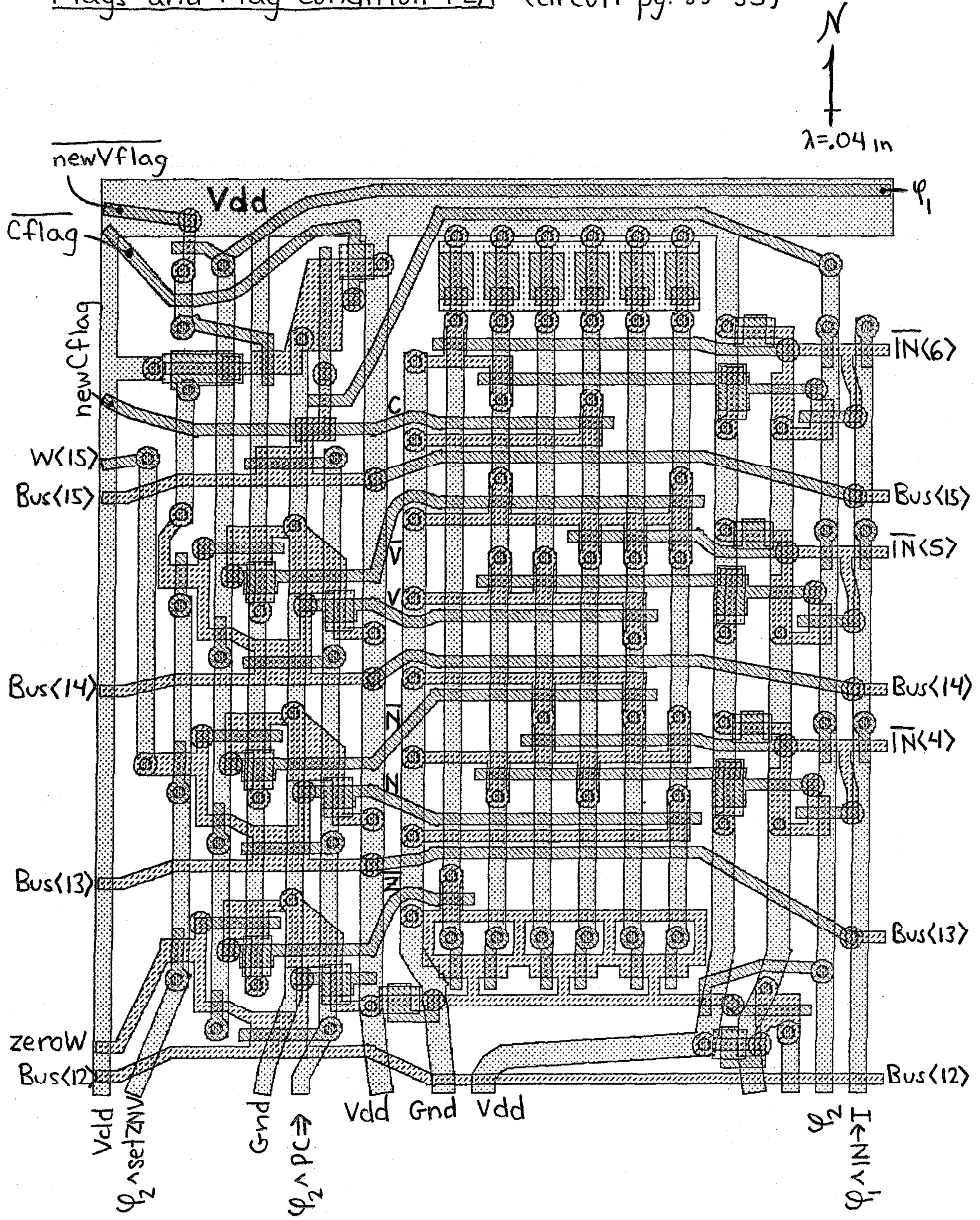
ALU Carry Chain (circuit pg.31)



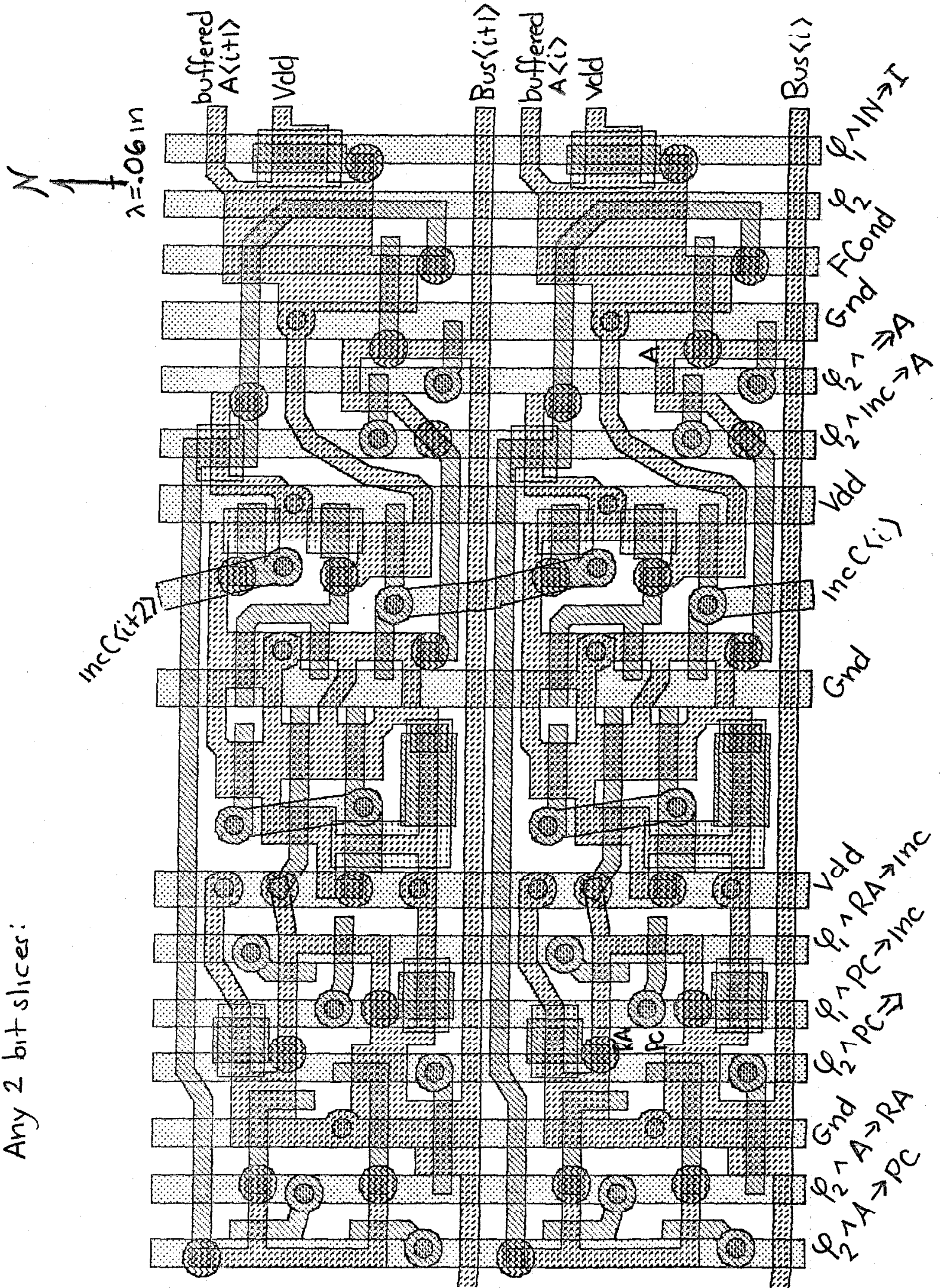
Non-restoring stage Restoring stage

(Snail indicates direction of carry propogation)

Flags and Flag Condition PLA (circuit pg. 33-35)



Address Section (circuit pg. 36)



Ports