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. 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.

\footnote{[Lute,Rabin,Soita&Spool83] is a short precursor to this thesis.}
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&Conway00]. 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 principal 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 $\varphi_1$ and $\varphi_2$, have minimum high times of roughly 60 nsec and 30 nsec, respectively. The secondary clocks, $\varphi_{1L}$ and $\varphi_{2L}$, 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.
5. Datapath

The datapath contains those parts of the processor that communicate over the internal data bus. The bus is 18 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 $\varphi_1$ of each cycle the bus is precharged and the ALU/shifter computes a new result. $\varphi_2$ 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 $\varphi_1$.

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 assumed 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, multiplexors, and other miscellaneous logic. Most of this complexity disappeared with the realization that a PLA could be efficiently programs 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. Programming this "folded" PLA is logically the same as programming 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 \( \Phi_a \) and \( \Phi_b \) 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, store, 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 processing, (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"). "J= *" 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 = 73]
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 DECIDE: .decode I= * INT=0 :: IN->I saveC RJ=> X Y D M RA++->A .go
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->X: .go I= 0 1 0 * * * * * :: MDAI II DF++->A W=> RK .decode

Processor timing in executing the macroinstruction:

<table>
<thead>
<tr>
<th>microcycle number</th>
<th>microcode word(s) being fetched</th>
<th>microcode word(s) controlling processor</th>
<th>memory address being computed</th>
<th>memory address</th>
<th>memory data available</th>
<th>ALU function and bus transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DECIDE: ...</td>
<td>RC+1 = 12 (immediate value)</td>
<td>11 (ADD instr.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>#J,K:</td>
<td>RA+1 (new refresh address)</td>
<td>12 (immediate value)</td>
<td>ADD instr. (latch into I register)</td>
<td>R1=&gt; X,Y,M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD: and ALU-&gt;X:</td>
<td>PC+1 = 13 (1st word of next instr.)</td>
<td>7 (immediate value)</td>
<td></td>
<td>7=&gt;X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DECIDE: and ALU-&gt;X:</td>
<td>PC+1 = 14 (immed. for next instr.)</td>
<td>13 (refresh data)</td>
<td>X+Y=&gt;W</td>
<td>W=&gt;R2</td>
<td></td>
</tr>
</tbody>
</table>

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 \( \varphi_2 \) 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 ("RI-> XY 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 "NOLU" 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=00"), the carry-in ("cin=0"), and the type of shift ("wshift", for "w 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=abc01000", since the OP code for ADD is 0100).

The "ALU->K:" word deposits the ALU/shifter output in register K ("= 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 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, organ-
ized 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 other-
wise 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 rea-
son 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
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 [Kingsley02], 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 EarI. This unusual layout style saved about 10% in area over 45 degree angle geometry, and about 50% 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
Stevie Rabin for quality control, verification, and memory section design
Chris Kingsley for Earl
Howard Dorby 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

[Browning&Seitz81]
Sally A Browning and C L Seitz
Communication in a Tree Machine
Computer Science, Caltech

[Bryant82]
Randy Bryant, Mike Schuster and Doug Whiting
MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual

[Bryant83]
Randal E Bryant
A Switch-Level Model and Stimulator for MOS Digital Systems

[Kingsley82]
Chris Kingsley
Earl: An Integrated Circuit Design Language

[Lutz,Rabin,Seitz&Speck83]
Chris Lutz, Etcwe Rabin, Chuck Seitz, and Don Speck
Design of the Mosaic Element
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
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 — Two's-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:

```
<table>
<thead>
<tr>
<th>MODE</th>
<th>OPCODE</th>
<th>R</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
followed optionally by one word of immediate value.

When K specifies a port:

```
<table>
<thead>
<tr>
<th>MODE</th>
<th>OPCODE</th>
<th>[Adv]Dir</th>
<th>pt</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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.
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.

<table>
<thead>
<tr>
<th>MODE</th>
<th>Dir</th>
<th>X</th>
<th>Y</th>
<th>Dest</th>
<th>Assembly language syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RJ</td>
<td>RK</td>
<td>RK</td>
<td></td>
<td>&lt;Mnemonic&gt; RJ {, Rk}</td>
</tr>
<tr>
<td>1</td>
<td>val</td>
<td>RJ</td>
<td>RK</td>
<td></td>
<td>&lt;Mnemonic&gt; #val , Rj {, Rk}</td>
</tr>
<tr>
<td>2</td>
<td>@RJ</td>
<td>RK</td>
<td>RK</td>
<td></td>
<td>&lt;Mnemonic&gt; @Rj {, Rk}</td>
</tr>
<tr>
<td>3</td>
<td>@val</td>
<td>RJ</td>
<td>RK</td>
<td></td>
<td>&lt;Mnemonic&gt; @#val , Rj {, Rk}</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>RJ</td>
<td>0</td>
<td>outport</td>
<td>&lt;Mnemonic&gt; RJ , Ppt</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>val</td>
<td>RJ</td>
<td>outport</td>
<td>&lt;Mnemonic&gt; #val {, Rj} , Ppt</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>inport</td>
<td>RJ</td>
<td></td>
<td>&lt;Mnemonic&gt; &lt;input&gt; , Rj</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>val</td>
<td>inport</td>
<td>RJ</td>
<td>&lt;Mnemonic&gt; #val , &lt;input&gt; , Rj</td>
</tr>
<tr>
<td>6</td>
<td>@RJ</td>
<td>RK</td>
<td>@RJ</td>
<td></td>
<td>&lt;Mnemonic&gt; M @Rj {, Rk}</td>
</tr>
<tr>
<td>7</td>
<td>@val</td>
<td>RJ</td>
<td>@val</td>
<td></td>
<td>&lt;Mnemonic&gt; M @#val {, Rj}</td>
</tr>
</tbody>
</table>

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

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

BRANCH CONDITIONS:

<table>
<thead>
<tr>
<th>K</th>
<th>Condition</th>
<th>Alternate &lt;Mnemonic&gt; (implies OPCODE=BRAT) (implies OPCODE=BKAR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00pt</td>
<td>Output port number pt Not Ready (i.e. no room in port to do output)</td>
<td>BONR</td>
</tr>
<tr>
<td>01pt</td>
<td>Input port number pt Not Ready (i.e. no word in input port to read)</td>
<td>BINR</td>
</tr>
<tr>
<td>1000</td>
<td>V [overflow]</td>
<td>BVS</td>
</tr>
<tr>
<td>1001</td>
<td>N [negative]</td>
<td>BNS</td>
</tr>
<tr>
<td>1010</td>
<td>~C [Carry = 0]</td>
<td>BCC or BLO</td>
</tr>
<tr>
<td>1011</td>
<td>N xor V [signed &lt; ]</td>
<td>BLT</td>
</tr>
<tr>
<td>1100</td>
<td>Z [zero]</td>
<td>BZS or BEQ</td>
</tr>
<tr>
<td>1101</td>
<td>Z or N [&lt;= zero]</td>
<td>BLEZ</td>
</tr>
<tr>
<td>1110</td>
<td>Z or ~C [unsigned &lt;=]</td>
<td>BLOS</td>
</tr>
<tr>
<td>1111</td>
<td>Z or (N xor V) [signed &lt;=]</td>
<td>BLE</td>
</tr>
</tbody>
</table>

When <OR <Mnemonic>> is BONR, BOR, BINR, or BIR; pt may be specified by writing "Ppt" in place of "Rk" field.
Version 14-Mar-83

! MACROS DEFINED IN ASSEMBLER:
! Cin=0 :: Cforce
! Cin=1 :: Cforce Cvall
! PC++->A :: PC->inc Add1 inc->A A->PC
! RA++->A :: RA->inc Add1 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= QA Cin=1 nosh W=>
! Y-1=> :: GP= A5 Cin=0 nosh W=>
! saveC :: rnb
! TESIX :: GP= OC Cin=0 nosh set.ZNV

! All alu outputs no xistors:
! NOALU :: GP= FF cforce setC saveC lsr ror asr rnb 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.

! 'WordO' 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=000 Word J,K,K: saveC PC->A RK=>Y .go
row .get I=001 Word #,J,K: saveC PC++->A IN=>X D .go
row .get I=*10 Word @J,K: saveC RJ=>A .get3
row .get3 I=010 Word @J,K,K: PC->A RK=>Y .get4
row .get3 I=110 Word @J,K,J: RK=>Y .get4
row .get I=*11 Word @#,J: saveC PC++->A IN=>X .get2
row .get2 Word @#,J2: X=>A .get3
row .get3 I=011 Word @#,J,K: PC->A .get4
row .get3 I=111 Word @#,J,K: .get4
row .get4 I=*1* Word any@: IN=>X D .go
row .get I=100 Word io;no#: saveC PC->A .io
row .get I=101 Word io;with#: saveC PC++->A IN=>X Y D .io
row .io I=10 PortC=1 Word I/O wait: X=>X Y D RA++->A .go+2
row .get2 I=10 Word wait2: PC->A saveC .io
row .io I=10* PortC=0 Word WillSend: RJ=>Y .io
row .io I=100 PortC=0 Word Get: Pt=>X D .io
row .io I=101 PortC=0 Word GetAdv: Pt=>Y .io
row .io I=101* PortC=0 Word GetAdv: Pt=>Y Advance .io

! 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=10*0 Word Alu->J: PC++->A NOALU W=>RJ .decode
row .go I=10*0 Word Alu->Out: PC++->A NOALU W=>Pt .decode
row .go I=11*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

! Rove to specify ALU functions
row .go I=**0000** Word0 mov: ALUONLY GP= 0C Cin=0 nosh setZNV
    Word1 com: ALUONLY GP= 03 Cin=0 nosh setZNV
row .go I=**0001** Word0 inc: ALUONLY GP= 0C Cin=1 nosh setZNV
    Word1 dec: ALUONLY GP= C3 Cin=0 nosh setZNV
row .go I=**0010** Word0 neg: ALUONLY GP= 03 Cin=1 nosh setZNV setC
    Word1 rnb: ALUONLY GP= 0C Cin=0 rnb setZNV
row .go I=**0011** Word0 ror: ALUONLY GP= 0C Cin=0 ror setZNV
    Word1 rol: ALUONLY GP= C0 nosh setZNV setC
row .go I=**0100** Word0 asr: ALUONLY GP= 0C Cin=0 asr setZNV
    Word1 lsr: ALUONLY GP= 0C Cin=0 lsr setZNV
row .go I=**0101** Word0 add: ALUONLY GP= 86 Cin=0 nosh setZNV setC
    Word1 addc: ALUONLY GP= 86 nosh setZNV setC
row .go I=**0110** Word0 sub: ALUONLY GP= 99 Cin=1 nosh setZNV setC
    Word1 and: ALUONLY GP= 08 Cin=0 nosh setZNV
row .go I=**0111** Word0 or: ALUONLY GP= OE 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=**1000** Word0 cmp: PC+->A GP= 42 Cin=1 nosh setZNV setC
    Word1 jump: X=> A A->PC .decode
                  .fetch

! Pushes and explicit stores
row .go I=**101** Word0 push: Y-1=> RK A .go2
    Word1 stox: Y=> A .go2
    Word0 push2: Write PC->A testX .fetch
    Word1 stox2: Write PC->A testX .fetch
row .go I=**1100** Word0 pushj: PC-> D .go2
    Word1 stof: Y=> D setZNV .go2
row .go2
    Word0 pushj2: Y-1=> RK A .go3
    Word1 stox2: X=> A .go3
row .go3
    Word0 pushj3: Write X=> A A->PC .fetch
    Word1 stoy3: Write PC->A .fetch

! Pops
row .go I=**1101** 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=**111** PortC=0 Word0 BraT_Pt=0: PC+-> A .decode
    Word1 BraF_Pt=0: X=> A A->PC .fetch
row .go I=**111** PortC=1 Word0 BraT_Pt=1: X=> A A->PC .fetch
    Word1 BraF_Pt=1: PC+-> A .decode
row .go I=**111** FlagC=0 Word0 BraT_Fl=0: PC+-> A .decode
    Word1 BraF_Fl=0: X=> A A->PC .fetch
row .go I=**111** FlagC=1 Word0 BraT_Fl=1: X=> A A->PC .fetch
    Word1 BraF_Fl=1: PC+-> A .decode
Pads (46):

Ground  Reset  Address Pad $\langle 0...11 \rangle$  scan_in Pad
$+\text{Vdd}$  Write  Data Pad $\langle 0...15 \rangle$  scan_out Pad
$\phi_1$  $\phi_a$  IP Pad $\langle 0...3 \rangle$
$\phi_2$  $\phi_b$  OP Pad $\langle 0...3 \rangle$

Controller Inputs (17):

Reset  I$\langle 6...15 \rangle$
FCond  FB$\langle 0...3 \rangle$ (feedback)
PCond

Controller Outputs (41):

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

Bus sources:  \( \overrightarrow{W} \overrightarrow{PC} \overrightarrow{IN} \overrightarrow{R} \overrightarrow{Pt} \)
Bus destinations:  \( \overrightarrow{X} \overrightarrow{Y} \overrightarrow{D} \overrightarrow{R} \overrightarrow{Pt} \overrightarrow{A} \) \( \Rightarrow F \text{ ver. B only} \)
ALU/shifter:  \( P_{\phi\phi}^* \ P_{\phi 1}^* \ P_{1\phi}^* \ P_{11}^* \)
\( G_{\phi 1}^* \ G_{1\phi}^* \ G_{11}^* \)
\( \text{rnib}^* \ \text{ror}^* \ \text{lslr}^* \ \text{asr}^* \ \text{nosh}^* \)
\( \text{setC}^* \ \text{set2NV} \ \text{Cforce}^* \ \overrightarrow{Cval}^* \)
Address section:  A$\Rightarrow$RA  A$\Rightarrow$PC  RA$\Rightarrow$inc*  PC$\Rightarrow$inc*  inc$\Rightarrow$A Add1
Misc.:  \( \overrightarrow{\text{advance}} \ \text{write} \ \overrightarrow{\text{IN} \Rightarrow I} \ \text{useJ} \ \text{FB} \langle 0...3 \rangle \)
Repeat for $i = 0$ to $15$:

- $\varphi_2^i \Rightarrow X$
- $\varphi_2^i \Rightarrow Y$
- $Y(i)$

**Bus $\langle i \rangle$**

**Propagate function block**

**Generate function block**

$P(i)$

$G(i)$

$\overline{aluC(i+1)}$

$\varphi_2$

$G(i)$

$P(i)$

$\overline{aluout(i)}$

$\overline{aluC(i)}$

Restoring: $(i = 0, 3, 6, 9, 12, 15)$

Not Restoring: (all other $i$)
Shifter

Bit 15:

\[ \text{aluout} <3>, \text{Cflag}, \text{aluout} <15> \]

\[ \varphi_i \wedge \text{rnib}, \varphi_i \wedge \text{ror}, \varphi_i \wedge \text{lsr}, \varphi_i \wedge \text{asr}, \varphi_i \wedge \text{nash} \]

\[ W<15>, \text{Bus}<15>, \text{zeroW} \]

Repeat for \( i = 0 \) to 14:

\[ \text{aluout} <i+4 \mod 16>, \text{aluout} <i+1>, \text{aluout} <i> \]

\[ \varphi_i \wedge \text{rnib}, \varphi_i \wedge \text{ror}, \varphi_i \wedge \text{lsr}, \varphi_i \wedge \text{nash} \]

\[ W<i>, \text{Bus}<i>, \text{zeroW} \]
Carry flag (and ALU Carry In)

\[ \varphi_2 \wedge \text{PC} \Rightarrow \]
\[ \text{Bus}(15) \]
\[ \text{newCflag} (= C) \]
\[ \varphi_1 \wedge \text{ror} \]
\[ \varphi_1 \wedge \text{lsr} \]
\[ \varphi_1 \wedge \text{asr} \]
\[ \text{ver.B only} \]

ALU Carry in

\[ \text{Cflag} \]
\[ \overline{\text{aluC}(\emptyset)} \]
\[ \text{Cval} \]
\[ \text{Cforce} \]
V, N, and Z flags

Ver. B only

Bus(14)

ϕ₂^F

ϕ₁

ϕ₂^PC

Bus(14)

V

ϕ₁

ϕ₂^PC

<alu C<16>

<alu C<15>

N

ϕ₁

ϕ₂^PC

Bus(13)

ϕ₂^F

N

W<15>

ϕ₁

ϕ₂^PC

Bus(12)

ϕ₂^F

ϕ₂^PC

Z

zeroW
Flag Condition PLA

<table>
<thead>
<tr>
<th>$I(6)$</th>
<th>$I(5)$</th>
<th>$I(4)$</th>
<th>preFCond</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$V$ [ $&lt; \emptyset$ ]</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>1</td>
<td>$N$ [ unsigned $&lt;$ ]</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\overline{C}$ [ signed $&lt;$ ]</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$N \oplus V$ [ signed $&lt;$ ]</td>
</tr>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>0</td>
<td>$Z$ [ $\leq \emptyset$ ]</td>
</tr>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>1</td>
<td>$Z \lor N$ [ $\leq \emptyset$ ]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$\emptyset$</td>
<td>$Z \lor \overline{C}$ [ unsigned $\leq$ ]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$Z \lor (N \oplus V)$ [ signed $\leq$ ]</td>
</tr>
</tbody>
</table>

\[
\text{preFCond} = ( I(6) \land Z \\
\lor ( I(6) \land I(5) \land I(4) ) \land V ) \\
\lor ( ( I(5) \land I(4) ) \lor N ) \\
\lor ( ( I(5) \land I(4) ) \land \overline{C} ) \\
\lor ( ( I(5) \land I(4) ) \lor N \lor \overline{V} ) \\
\lor ( ( I(5) \land I(4) ) \land \overline{N} \land V )
\]

Use PLA to compute NOR-NOR form of preFCond

\[
\psi_2
\]

(preFCond) \rightarrow \neg \rightarrow \text{FCond} (\text{to controller})
Address Section

Repeat for $i = 0$ to 11:

![Circuit Diagram]

- $\phi_2 \land A \rightarrow RA$
- $\phi_1 \land RA \rightarrow inc$
- $\phi_2 \land inc \rightarrow A$
- $A \rightarrow inc$
- $Bus(i)$
- $\phi_2 \land PC$
- $\phi_1 \land PC \rightarrow inc$
- $\phi_2 \land PC \rightarrow inc$
- $\phi_2 \land A$

- $Inc(i)$
- $IncC(i)$
- $IncC(i+1)$
- $IncC(\phi) \equiv Add1$
- $Incin(i)$
- $Incout(i)$

Address

Pod $\langle i \rangle$
Register Array

Repeat for \( n = 0 \) to 15:

\[
\{ \\
\text{Repeat for } i = 0 \text{ to } 15 \\
\}
\]

Place dotted transistors to pull down when

\[
n \neq (S(0) + 2 S(1) + 4 S(2) + 8 S(3))
\]
S (Register Select) Generation

\[ S(3) \quad S(2) \quad S(1) \quad S(\varnothing) \quad \overline{S}(3) \quad \overline{S}(2) \quad \overline{S}(1) \quad S(\varnothing) \]

\[ I(7) \quad I(6) \quad I(5) \quad I(4) \quad I(3) \quad I(2) \quad I(1) \quad I(\varnothing) \]

Use J \quad \{ I(4), I(5), I(6) \} \quad \text{to Ports}
Memory Data Interface,
Bus Precharge
and Instruction Register

Repeat for $i = 0$ to 15:

For $i = 4, 5, 6$:
near flag condition PLA
For $i = 0 \ldots 7$:
near register select
For $i = 6 \ldots 15$:
equivalent circuit built into controller
(Note duplications for some bits)

*I<i>*

not needed in version B
Output Ports

Repeat for $n = 0$ to $3$:

$\varphi_i \land \text{OPshift}[n]$  

$\neg \varphi_i$ \land \text{OPshift}[n]

$\varphi_2$ \lor \text{OPload}[n]

$\text{empty}[n]$

$\text{Bus}(i)$

$\text{Pulldown for } i = 0 \text{ to } 13$ only

Repeat for $i = 0$ to $15$

$\varphi_1 \land \text{OPshift}[n]$

$\neg \varphi_2$

$\neg \varphi_1$ \land \text{OPshift}[n]

$\text{OP Pa}[n] \land \text{OP Pa}[n](-1)$

$\varphi_2$ \lor \text{OPload}[n]

$\text{empty}[n]$

$\text{Pulldown for } i = 0 \text{ to } 13$ only
Output Ports pg. 2 of 2

Repeat for \( n = 0 \) to 3:

\[
\begin{align*}
\varphi_1 \land \overline{OP_{shift}[n]} & \quad \varphi_1 \land \overline{OP_{shift}[n]} \\
OP_{Pad}[n] & \quad \varphi_2 \\
\overline{OP_{a}[n]} & \quad \text{OutCond}
\end{align*}
\]

\[
\begin{align*}
\varphi_2 \land OP_{load}[n] & \quad \text{empty}[n] \\
+ & \quad \Rightarrow Pt \\
\overline{I\langle 6 \rangle} & \quad \text{I\langle 6 \rangle} \\
\overline{I\langle 6 \rangle} & \quad \text{I\langle 5 \rangle} \\
\text{I\langle 5 \rangle} & \quad \text{I\langle 6 \rangle} \\
\text{I\langle 6 \rangle} & \quad \text{I\langle 4 \rangle} \\
\overline{I\langle 4 \rangle} & \quad \text{I\langle 4 \rangle}
\end{align*}
\]

Place dotted transistors to pull down when
\[ n \neq \text{I\langle 4 \rangle} + 2 \text{I\langle 5 \rangle} \]
Input Ports pg. 1 of 2
Repeat for \( n = \emptyset \) to 4:

- \( \varphi_2 \) \( \land \) IPfull[n]
- \( \varphi_1 \) \( \land \) IPfull[n]

- \( \varphi_2 \) \( \land \) IPread[n]
  - for \( i = \emptyset \) to 15 only
- \( \varphi_1 \) \( \land \) IPshift[n]
- \( \varphi_2 \) \( \land \) IPshift[n]

Repeat for \( i = \emptyset \) to 16

- IP Pad[n]
- IP full[n]
- \( \varphi_2 \) \( \land \) IPshift[n]
Input Ports pg 2 of 2 (and Port Condition) Mosaic ver.A'11.8'11

Repeat for $\phi = \emptyset$ to 3:

$\phi_1^{\text{advance}}[n]$

$\phi_2^{\text{IPread}}[n]$

$\phi_2^{\text{IPfull}}[n]$

Place dotted transistors to pull down when $n \neq I(4) + 2I(5)$
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 — Two's-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:

<table>
<thead>
<tr>
<th>K</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

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

When K specifies a port:

<table>
<thead>
<tr>
<th>Adv</th>
<th>Dir</th>
<th>pt</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>MSOURCE</th>
<th>MDEST</th>
<th>K</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>MSOURCE</th>
<th>X</th>
<th>&lt;source&gt;</th>
<th>Time(MSOURCE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Rj</td>
<td>Rj</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>@Rj</td>
<td>@Rj</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>@Rj++</td>
<td>@Rj++</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>@Rj+val</td>
<td>@Rj+val</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>val</td>
<td>#val</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>@val</td>
<td>@#val</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Input Port pt</td>
<td>Ppt+</td>
<td>[advance] 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ppt=</td>
<td>[no advance] 1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MDEST</th>
<th>effect</th>
<th>&lt;destination&gt;</th>
<th>Time(MDEST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X -&gt; Rj</td>
<td>Rj</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X -&gt; @Rj</td>
<td>@Rj</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>X -&gt; @Rj++</td>
<td>@Rj++</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>X -&gt; @Rj+val</td>
<td>@Rj+val</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>X -&gt; @--Rj</td>
<td>@--Rj</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>X -&gt; @val</td>
<td>@#val</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>X -&gt; Output Port pt</td>
<td>Ppt</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>X -&gt; nowhere</td>
<td>.</td>
<td>0</td>
</tr>
</tbody>
</table>

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.
ARITHMETIC AND BRANCH INSTRUCTION MODES:

<table>
<thead>
<tr>
<th>MODE</th>
<th>OP1</th>
<th>OP2</th>
<th>OP3</th>
<th>K</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>MODE</th>
<th>X</th>
<th>Y</th>
<th>Dest</th>
<th>Assembly language syntax (when Rk not specified, k=j)</th>
<th>Time(MODE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Rj</td>
<td>Rk</td>
<td>Rk</td>
<td>&lt;OP Mnemonic&gt; Rj {j, Rk}</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>val</td>
<td>Rj</td>
<td>Rk</td>
<td>&lt;OP Mnemonic&gt; #val, Rj {j, Rk}</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>@Rj</td>
<td>Rk</td>
<td>Rk</td>
<td>&lt;OP Mnemonic&gt; @Rj {j, Rk}</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>@val</td>
<td>Rj</td>
<td>Rk</td>
<td>&lt;OP Mnemonic&gt; @#val, Rj {j, Rk}</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>@Rj</td>
<td>Rk</td>
<td>@Rj</td>
<td>&lt;OP Mnemonic&gt; M @Rj {j, Rk}</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>@val</td>
<td>Rj</td>
<td>@val</td>
<td>&lt;OP Mnemonic&gt; M @#val {j, Rj}</td>
<td>4</td>
</tr>
</tbody>
</table>

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

BRANCH CONDITIONS:

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

<table>
<thead>
<tr>
<th>OP</th>
<th>Instruction</th>
<th>&lt;OP Mnemonic&gt;</th>
<th>Effect</th>
<th>C flag modified?</th>
<th>TIME(OP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INCrement</td>
<td>INC</td>
<td>X + 1</td>
<td>→ Dest</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>DECrement</td>
<td>DEC</td>
<td>X - 1</td>
<td>→ Dest</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Arithmetic Shift Right</td>
<td>ASR</td>
<td>X&lt;15&gt;</td>
<td>X</td>
<td>→ Dest</td>
</tr>
<tr>
<td>3</td>
<td>Arithmetic Shift Left</td>
<td>ASL</td>
<td>X + X</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>ROR</td>
<td>ROR</td>
<td>C</td>
<td>X</td>
<td>→ Dest</td>
</tr>
<tr>
<td>5</td>
<td>Rotate Left</td>
<td>ROL</td>
<td>X + X + C</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>Logical Shift Right</td>
<td>LSR</td>
<td>0</td>
<td>X</td>
<td>→ Dest</td>
</tr>
<tr>
<td>7</td>
<td>Rotate Nibble Right</td>
<td>RNR</td>
<td>X&lt;3:0&gt;</td>
<td>X15:4</td>
<td>→ Dest</td>
</tr>
<tr>
<td>8</td>
<td>ADD</td>
<td>ADD</td>
<td>X + Y</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>ADD with Carry</td>
<td>ADDC</td>
<td>X + Y + C</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>A</td>
<td>SUBtract</td>
<td>SUB</td>
<td>Y - X</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>SUBtract with Carry</td>
<td>SUBC</td>
<td>Y - X - 1 + C</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>SUBtract Negate</td>
<td>SUBN</td>
<td>X - Y</td>
<td>→ Dest</td>
<td>γco</td>
</tr>
<tr>
<td>D</td>
<td>SUBtract Negate with Carry</td>
<td>SUBNC</td>
<td>X - Y - 1 + C</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>NEGate</td>
<td>NEG</td>
<td>~X</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>F</td>
<td>INCrement with Carry</td>
<td>INCC</td>
<td>X + C</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>10</td>
<td>bitwi se COMplement</td>
<td>COM</td>
<td>~X</td>
<td>→ Dest</td>
<td>no</td>
</tr>
<tr>
<td>11</td>
<td>bitwise AND</td>
<td>AND</td>
<td>X and Y</td>
<td>→ Dest</td>
<td>no</td>
</tr>
<tr>
<td>12</td>
<td>bitwise OR</td>
<td>OR</td>
<td>X or Y</td>
<td>→ Dest</td>
<td>no</td>
</tr>
<tr>
<td>13</td>
<td>bitwise eXclusive OR</td>
<td>XOR</td>
<td>X exor Y</td>
<td>→ Dest</td>
<td>no</td>
</tr>
<tr>
<td>14</td>
<td>CoMPare</td>
<td>CMP</td>
<td>X - Y</td>
<td>→ Dest</td>
<td>yes</td>
</tr>
<tr>
<td>15</td>
<td>BIT Test</td>
<td>BITT</td>
<td>X and Y</td>
<td>→ Dest</td>
<td>no</td>
</tr>
<tr>
<td>16</td>
<td>unsigned MULtiply</td>
<td>MUL</td>
<td>high word(X*Y)</td>
<td>→ RJ</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>low word(X*Y)</td>
<td>→ RK</td>
<td>modify Z, N, V based on high word</td>
</tr>
<tr>
<td>17</td>
<td>undefined</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BRANCH INSTRUCTIONS:

<table>
<thead>
<tr>
<th>OP</th>
<th>Instruction</th>
<th>&lt;OP Mnemonic&gt;</th>
<th>Effect</th>
<th>TIME(OP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>JUMP</td>
<td>JUMP</td>
<td>X → PC</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>Jump and ReStore flags</td>
<td>JRST</td>
<td>X → FPC</td>
<td>1</td>
</tr>
<tr>
<td>1A</td>
<td>POP Jump</td>
<td>POPJ</td>
<td>@Rk++ → PC</td>
<td>3</td>
</tr>
<tr>
<td>1B</td>
<td>POP Jump and Restore flags</td>
<td>POPJR</td>
<td>@Rk++ → FPC</td>
<td>3</td>
</tr>
<tr>
<td>1C</td>
<td>Branch True</td>
<td>BRAT</td>
<td>X → PC if Condition k true</td>
<td>0 or 1 *</td>
</tr>
<tr>
<td>1D</td>
<td>Branch False</td>
<td>BRAF</td>
<td>X → PC if Condition k false</td>
<td>0 or 1 *</td>
</tr>
<tr>
<td>1E</td>
<td>PUSH Jump</td>
<td>PUSHJ</td>
<td>Y-1 → Rk; FPC → @Rk; X → PC</td>
<td>4</td>
</tr>
<tr>
<td>IF</td>
<td>undefined</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* 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. D microcode  page 1 of 7

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

! MACROS:
DEF Cin=0  Cforce
DEF Cin=1  Cforce Cvall
DEF Ru->A  Rn->1nc Addl inc->A A->Rn
DEF RA->A  RA->1nc Addl inc->A A->RA
DEF PC->A  PC->1nc inc->A A->PC
DEF RJ=>  useJ R=>
DEF RK=>  R=>
DEF RJ  useJ R
DEF RK  R
DEF X=>  GP= OC Cin=0 nosh W=>
DEF Y=>  GP= QA Cin=0 nosh W=>
DEF X+Y=>  GP= 86 Cin=0 nosh W=>
DEF Y+1=>  GP= QA Cin=1 nosh W=>
DEF Y-1=>  GP= A5 Cin=0 nosh W=>
DEF X+1=>  GP= OC Cin=1 nosh W=>
DEF X-1=>  GP= C3 Cin=0 nosh W=>
DEF |-1=>  GP= OF Cin=0 nosh W=>
DEF 0=>  GP= 00 Cin=0 nosh W=>

! rnb refreshes the carry flag. This is done on the cycle
! after DISPATCH: of every instruction.
DEF saveC  rnb
DEF test:X  GP= OC Cin=0 nosh setZNV

! FEEDBACK MNEMONICS:
DEF .reset2  FB=  0  0  0  0
DEF .reset3  FB=  0  0  0  1
DEF .reset4  FB=  0  0  1  0
DEF .interrupt2  FB=  0  0  0  1
DEF .interrupt3  FB=  0  0  1  0
DEF .interrupt4  FB=  0  0  1  0
DEF .interrupt5  FB=  0  0  1  0
DEF .interrupt6  FB=  0  0  1  1
DEF .refetch  FB=  0  1  0  0
DEF .fetch  FB=  0  1  0  1
DEF .decode  FB=  0  1  0  1
DEF .get  FB=  0  1  1  0
DEF .get2  FB=  0  1  1  0
DEF .get3  FB=  0  1  1  0
DEF .get4  FB=  0  1  1  1
DEF .go  FB=  1  0  0  0
DEF .go2  FB=  1  0  0  1
DEF .go3  FB=  1  0  1  0
DEF .mov  FB=  1  0  1  1
DEF .mov2  FB=  1  0  1  0
DEF .mov3  FB=  1  0  1  0
DEF .store  FB=  1  0  1  0
DEF .RJ4  FB=  1  0  1  1
DEF .PC-2  FB=  1  1  0  0
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
- 'Sout' 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=5' 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
  - 'rnh' Rotate nibble
  - 'nsh' 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
  - 'Addi' 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 reset5: .reset5 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. D microcode

I REFETCH ON FAILED I/O IN MOVES

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

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

word KJ=>: .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

I 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 hadPt=>: .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 :: D=> X D Advance .mov
word iofailed: .get2 PortC=1 I= 0 0 1 1 0 :: PC=> X .refetch
word O=>: .get I= 0 0 1 1 1 :: O=> X D PC=>A .mov
! Mosaic ver. 8 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: .movv 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  .movv2
word ->@R+2: .movv2 :: Write X+1-> R PC->A  .fetch
word ->@R+:  .mov I= 0 0 * * * 0 1 1  :: testX R-> X  .mov
word ->@R+:  .movv2 :: IN-> Y  PC++->A  .movv3
word ->@R+:  .movv3 :: X+Y-> A  .store
word ->@R:  .mov I= 0 0 * * * 1 0 0  :: testX R-> X  .movv2
word ->@R:  .movv2 :: 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  .movv2
word ->@#:  .movv2 :: 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 ->Pt(wait: 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
! ARITHMETIC AND BRANCH SOURCES

word J,K,K: .get I= 0 1 0 :: PC->A RK=> Y .go
word @J,K,K: .get I= 0 1 1 :: PC++->A IN=> X .go
word @J,K,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,K: .get2 I= 1 1 0 :: KK=> Y .get3
word @J,J,: .get I= 1 * 1 :: IN=> A .get2
word @J,J,J: .get2 I= 1 0 1 :: PC++->A .get3
word @J,J,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= OC Cin=1 nosh setZNV
word dec: .go I= ** 0 0 0 0 1 :: ALUONLY GP= C3 Cin=0 nosh setZNV
word asl: .go I= ** 0 0 0 1 0 :: ALUONLY GP= OC 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= OC Cin=0 ror setZNV
word rol: .go I= ** 0 0 1 0 1 :: ALUONLY GP= C0 nosh setZNV setC
word ror: .go I= ** 0 0 1 1 0 :: ALUONLY GP= OC Cin=0 lsr setZNV
word ror: .go I= ** 0 0 1 1 1 :: ALUONLY GP= OC Cin=0 rrf 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 inc: .go I= ** 0 1 1 1 1 :: ALUONLY GP= m Cin=0 nosh setZNV setC
word com: .go I= ** 1 0 0 0 0 :: ALUONLY GP= 03 Cin=0 nosh setZNV setC
word and: .go I= ** 1 0 0 0 1 :: ALUONLY GP= 06 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->Q: .go I= 1 1 * 0 * * * :: NOALU W=> D .store
word ALU->Q: .go I= 1 1 * 1 0 0 * * :: NOALU W=> D .store
word ALU->Q: .store I= 1 1 0 :: Write PC->A .fetch
word ALU->Q: .store I= 1 1 1 :: Write PC++->A .fetch
! Mosaic ver. B microcode  page 7 of 7

! SPECIAL ARITHMETICS: MULITPLY, ETC.

word cap:  .go  I=***10100  ::  PC++->A GP= 49 Cin=1 nosh setZNV setC  
            .decode

word bitt:  .go  I=***10101  ::  PC++->A GP= 08 Cin=0 nosh setZNV

word mu1:   .go  I=***10111*  ::  Mshift O->Y SRin=1  .go2
            .decode

word mulu:  .go2  SKout=U Mout=U  ::  Mshift Y->Yshift RA++->A  .go2

word mul:   .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=***11000  ::  X->A A->PC  .fetch

word jrst:   .go  I=***11001  ::  X->F A->PC  .fetch

word popj:    .go  I=***1101*  ::  RK->Y A  .go2

word popj2:  .go2  ::  Y+1->RK  .go3

word popj3:   .go3  I=***11010  ::  IN->A A->PC  .fetch

word popjr:  .go3  I=***11011  ::  IN->F A A->PC  .fetch

word ->PC(P=0): .go  PortC=0 I=***111000  ::  PC++->A  .decode

word ->PC(P=1): .go  PortC=1 I=***111000  ::  X->A A->PC  .fetch

word ->PC(F=0): .go  FlagC=0 I=***111001  ::  PC++->A  .decode

word ->PC(F=1): .go  FlagC=1 I=***111001  ::  X->A A->PC  .fetch

word ->PC(F=0): .go  FlagC=0 I=***111011  ::  PC++->A  .decode

word ->PC(F=1): .go  FlagC=1 I=***111011  ::  X->A A->PC  .fetch

word ->PushJ:  .go  I=***1111*  ::  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

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:
\[ \varphi_{1L} \quad \varphi_{2L} \quad \text{INT Pad} \]

Additional Controller Inputs:
\[ \text{INTff} \quad \text{SRout} \quad \text{Mout} \quad \text{FB}<4> \]

Additional Controller Outputs:
* indicates type "raw." All others type "C1"
\[ M \Rightarrow \]
\[ \Rightarrow M \quad \Rightarrow \text{Yshift} \quad \Rightarrow F \]
\[ \text{Mshift1} \ast \]
\[ \text{SRin} \ast \]
\[ \text{INT}<\emptyset \]
\[ \text{FB}<4> \]
Controller
(Replaces Mosaic ver. A controller)

 Relay

\[ \varphi_{2L} \land \text{IN} \rightarrow \text{I when input is } \text{IN} \langle \text{I5...I6} \rangle \]
\[ \varphi_{2L} \text{ otherwise} \]

Connect scan-in's to scan-out's. First scan-in to scan-in Pad. Last scan-out to scan-out Pad.

(type "C1"

select output type per output

scan-out)

Repeat for each implicant

Repeat for each input

Repeat for each output
**SR and M Registers**

**SR**
Shift Register for counting multiply steps

\[ SR^{(i)} = SR_{in} \]

Repeat for
\[ i = 0 \text{ to } 15 \]

\[ SR^{(i)} = SR_{out} \]

**M**
Multiplier/Product Register

\[ M^{(i)} = Y^{-1} \]

Repeat for
\[ i = 0 \text{ to } 15 \]

\[ M^{(i)} = M_{out} \]
**Y register** (and Interrupt Flip Flop)

- **Y** (Replaces Y register of version A)

![Diagram of Y register]

- Repeat for i = 0 to 14

- **Interrupt flip flop**

![Diagram of interrupt flip flop]
ALU input latches and function blocks (circuit pg. 31)
ALU Carry Chain (circuit pg 31)

Non-restoring stage (Snail indicates direction of carry propagation)
Flags and Flag Condition PLA (circuit pg. 37-35)
Any 2 b+ slices:

Address Section (circuit pg 36)