The M1 Processor Circuit for Sigma16

John T. O'Donnell

October 7, 2021

Contents

1 Quick start: tl;dr

```
$ cd circuits
$ ghci
ghci> :load M1/Run
ghci> :main programs/Add
Sigma16.M1> run
Sigma16.M1> quit
ghci> :quit
```

2 Introduction

Sigma16 is a computer architecture designed for research and teaching in computer systems. M1 is a CPU: it is a digital circuit that executes the Core subset of Sigma16.

M1 is a synchronous circuit designed at the level of logic gates and flip flops. It is specified using the Hydra hardware description language, which describes precisely the components and how they are connected, and which can simulate the circuit.

The Circuit directory contains the M1 circuit, which is organized as several subsystems. The Run module defines a simulation driver; this is a command line text interface that provides human readable inputs and outputs. The ReadObj module is used to boot the system: it reads an object code file produced by the Sigma16 assembler and automatically generates the inputs required to make the circuit boot the program.

3 Running the system

3.1 Installation

- Install ghc
- Install Hydra

3.2 Configuration

There is a directory `circuits/programs` which contains several Sigma16 programs. The examples below use these programs, and you can put your own programs there too. If you do so, you can skip this section.

However, you may wish to keep the circuits directory and your Sigma16 programs directory in different locations in your file system. If you do that, you may end up with long file paths to your object code, and this can make it hard to type in the command to load a program. The M1 driver has a facility to make that easier.

If the file `circuits/fileprefix.txt` exists, then the text in its first line is attached to the beginning of the file specified in the argument to the `:main` command. For example, suppose the full path to your object code file is `/some/long/path/to/your/code/MyProgram.obj.txt`. Then you can put `/some/long/path/to/your/code/` into `fileprefix.txt`, and then run the program with just `:main MyProgram`.

If you don't want anything prefixed to your file arguments, either set fileprefix.txt to a blank line, or simply delete it.

3.3 An example program

Before examining the circuit, let's begin by taking a Sigma16 assembly language program and running it on the circuit. This is `circuits/programs/Add.asm.txt`:

```
; Add: a minimal program that adds two integer variables
; This file is part of Sigma16; see README and https://jtod.github.io/home/Sigma16/

; Execution starts at location 0, where the first instruction will be
; placed when the program is executed.

      load   R1,x[R0]    ; R1 := x
      load   R2,y[R0]    ; R2 := y
      add    R3,R1,R2    ; R3 := x + y
      store  R3,z[R0]    ; z := x + y
      trap   R0,R0,R0    ; terminate

; Expected result: z = x + y = 23 + 14 = 37 (hex 0025)

; Static variables are placed in memory after the program

x     data  23
y     data  14
z     data   0
```

3.4 Assemble the program

Computers don't run assembly language, they run machine language. So we need to translate the program. There are several ways to do this:

- *Command.* If you have installed the Sigma16 tools on youc computer, enter this command: `sigma16 assemble programs/Add`
- *Sigma16 app.* Launch Sigma16, load the example program, assemble it, and save the object code in a file. The file must be named `Add.obj.txt`
- *Assemble it by hand.* It's important to know *how* to assemble a program by hand, and it's worth doing one or two times. But once you understand how to translate from assembly to machine language, it's better to use the software tools.

3.5 Run the machine code on the M1 circuit

Now you can run the machine language program `Add.obj.txt` on the circuit:

- *Enter the circuits directory.* `$ cd Sigma16/src/circuits` The ghci command must be executed in this directory.
- *Launch Haskell.* `$ ghci`
- *Launch Hydra and M1.* `ghci> :load M1/Run`
- *Load the machine language program.* `ghci> :main programs/Add`
- *Boot the program and run it on the circuit.* `Sigma16.M1> run`
- *Leave M1 driver.* `Sigma16.M1> quit`
- *Leave ghci.* `ghci> :quit`

3.6 Breakpoints

The machine may exectue many clock cycles before it reaches a state that you're interested in. For example, if you want to examine exactly how the circuit executes a jal instruction, you need to get through the boot process and then all the instructions that execute before the jal. This can take a long time, and you may have to do it repeatedly.

The M1 simulation driver provides *breakpoints* which alleviate this problem. The idea is that you specify that a bit signal of interest is a breakpoint.

Then when you enter a run command, it will perform clock cycles repeatedly until the breakpoint signal becomes 1. At that point the simulation stops and you can examine the machine state in detail, and single step (or run) from that point on.

several t When you start the system (e.g. :main Arrays/ArrayMax) it will take a log of clock cycles to boot the machine language program. If there are n words of instructions and data, it will take n clock cycles. It's useful to enter these commands:

```
break reset
run
```

This will run the simulation without stopping, until the reset signal becomes 1, and then it will stop. That way you can start single stepping through the program, but don't have to single step through the boot.

3.7 Reference: prompts and commands

Prompts

```
$             is the bash shell prompt
 ghci:        is the ghci prompt
 Sigma16.M1>  is the circuit prompt
```

Useful ghci commands (see ghc User Guide for full documentationO

```
$           is the bash shell prompt
:r          reload after editing any of the code
:q          exit ghci, go back to shell
^C          stop and return to ghci prompt
uparrow     repeat previous command
```

4 Datapath

The datapath of a processor contains the registers, the circuits that perform calculations (ALU and functional units), and the buses that connect them. All of these subsystems take control inputs that determine their behavior. Those control signals are generated by the control system, which is not part of the datapath.

4.1 ALU

The ALU (arithmetic and logic unit) is a combinational circuit that performs calculations which can be completed efficiently in one clock cycle. The ALU performs integer additions, subtractions, comparisons, and the the like. However, more complex operations, such as multiplication, division, and all floating point operations, require more than one clock cycle. They also require some additional state (registers), and are typically performed in functional units.

The ALU calculates a function of word inputs x and y (which are usually the contents of two registers) and cc (the contents of R15). It produces a word output r, which is a numeric result (typically loaded into the destination register), and a comparison result ccnew which is the new value to be loaded into the condition code register. The ALU performs addition, subtraction, negation, increment, and comparision. The function is determined by two control signals (alua, alub).

Furthermore, the ALU receives ir_d as an input and uses it to output cond, which is the bit in cc indexed by ir_d. This is used by jumpc0 and jumpc1.

Control inputs:

- alua, alub

Data inputs:

- x
- y
- cc
- ir_d

Data outputs:

- r = function (a,b) x y cc
- ccnew = compare x y cc
- cond = $\mathrm{R15.ir}_\mathrm{d}$ (use ir_d to select bit from condition code)

The data output r is the result of an arithmetic operation which is determinted by the control inputs:

op = (alua, alub, aluc)

a b c	r
0 0 0	x+y
0 0 1	x-y
0 1 0	-x
0 1 1	x+1
1 0 0	cmp

The control algorithm defines all control signals, regardless of what operation is being performed.

Sigma16 defines the following condition code flags.

bit index	Relation	Symbol
0	$>$ Int	g
1	$>$ Nat	G
2	$=$	$=$
3	$<$ Nat	L
4	$<$ Int	$<$
5	Int overflow	v
6	Nat overflow	V
7	Carry	C
8	Stack overflow	S
9	Stack underflow	s

```
alu n (alua,alub,aluc) x y cc d = (sum, ccnew, cond)
  where
    negating = and2 (inv alua) (xor2 alub aluc)  -- alu abc = 001 or 010
    arith = inv alua  -- doing arithmetic operation, alu abc one of 000 001 010 100
    comparing = and3 alua (inv alub) (inv aluc)  -- doing comparison, alu abc = 100
    wzero = fanout n zero
    wone = boolword n one
-- prepare inputs to adder
    x' = mux2w (alub,aluc) x x wzero x
    y' = mux2w (alub,aluc) y (invw y) (invw x) wone
    xy = bitslice2 x' y'
    (carry,sum) = rippleAdd negating xy
-- binary comparison
    (lt,eq,gt) = rippleCmp xy
-- two's complement comparison
    lt_tc = mux2 (xy!!0) lt zero one lt
    eq_tc = eq
    gt_tc = mux2 (xy!!0) gt one zero gt
```

```
-- carry and overflow
    natovfl = carry
    intovfl = zero -- ????
-- condition code flags
    fcarry = and2 carry (inv comparing)
    fnatovfl = and2 natovfl (inv comparing)
    fintovfl = and2 intovfl (inv comparing)
    ccnew = [ zero,   zero,     zero,     zero,   -- bit 15 14 13 12
              zero,   zero,     zero,     zero,   -- bit 11 10  9  8
              fcarry, fnatovfl, fintovfl, lt_tc,  -- bit  7  6  5  4
              lt,     eq,       gt,       gt_tc   -- bit  3  2  1  0
            ]
-- conditional bit controls conditional jumps
    cond = indexbit d cc
```

– indexbitcs xs: select element at index cs from xs, where index 0 is – least significant position; require that length xs = 2 ˆ length cs

indexbit :: Bit a => [a] -> [a] -> a indexbit [] [x] = x indexbit (c:cs) xs = mux1 c (indexbit cs (drop i xs)) (indexbit cs (take i xs)) where i = 2 ˆ length cs

– mux4 cs xs: select element at index cs from xs; require that length – xs = 2 ˆ length cs

muxw :: Bit a => [a] -> [a] -> a muxw [] [x] = x muxw (c:cs) xs = mux1 c (muxw cs (take i xs)) (muxw cs (drop i xs)) where i = 2 ˆ length cs

4.1.1 Condition codes

```
--    comparing = and3 a b (or2 c d)
--    comp_bool = mux2 (c,d) zero lt_tc eq_tc gt_tc
--    comp_word = boolword n comp_bool
--    z = mux1w comparing sum comp_word
```

4.2 Basic register file

```
regfile1
  :: CBit a   -- sequential circuit with signal type a
  => Int      -- k is number of reg address bits, with 2^k registers
  -> a        -- ld.  if ld then reg[d] := x
  -> [a]      -- d is destination address, register to load
  -> [a]      -- sa is source address a, first operand to fetch
  -> [a]      -- sb is source address b, second operand to fetch
  -> a        -- x is data to load into reg[d] if ld=1
  -> (a,a)    -- (reg[sa], reg[sb]) is the readouts of two registers

regfile1 k ld d sa sb x
  | k==0 = (r,r)
  | k>0  = (a,b)
  where
    r = reg1 ld x
    (a0,b0) = regfile1 (k-1) ld0 ds sas sbs x
    (a1,b1) = regfile1 (k-1) ld1 ds sas sbs x
    (ld0,ld1) = demux1 d1 ld
    a = mux1 sa1 a0 a1
    b = mux1 sb1 b0 b1
    (d1:ds) = d
    (sa1:sas) = sa
    (sb1:sbs) = sb
```

The regfile1 circuit has 2^k words, each consisting of 1 bit. A full register file with n-bit words simply consists of n copies of the regfile1 circuit.

```
regfile :: CBit a => Int -> Int
  -> a -> [a] -> [a] -> [a] -> [a] -> ([a],[a])
regfile n k ld d sa sb x =
   unbitslice2 [regfile1 k ld d sa sb (x!!i)  | i <- [0..n-1]]
```

4.3 Handling R0 and R15

The basic register file treats all registers the same. However, Sigma16 treats R0 and R15 as special cases:

- R0 is always 0. It is legal to load another value into it, but any readout of R0 will yield 0.
- R15 is the condition code. It holds the result of the cmp instruction. Furthermore, arithmetic instructions (which place their result in the register specified by `Reg[ir_d]`) also set some flags in the condition code indicating overflow and other conditions. The conditional jump instructions automatically fetch bits from R15.

There are several ways the circuit could handle R0. M1 uses the best approach: R0 does not actully have any flip flops. A load into R0 is simply discarded, and the combinational logic for fetching a register produces the value 0 when R0 is fetched. This approach is relatively simple to implement, and the resulting circuit consumes less power and requires less chip area than alternatives that use flip flops to hold the irrelevant state of R0.

R15, which holds the condition code, is more complex. There are two reasons:

- The machine may need to read out both R15 and some other register at the same time.
- The machine may need to load a new value into both R15 and some other register at the same time.

Both of these requirements cannot be satisfied with the basic register file circuit. Instead, it would be necessary to use two clock cycles.

It would be possible to handle the condition code by using two clock cycles for arithmetic instructionss, one to put the result into the destination register and another to put the condition code into R15. However, add instructions are frequently executed, and this would give an unacceptable slowdown. (The M1 circuit is designed to be as simple as possible, but more advanced circuits should be possible with the architecture.)

It is critically important to be able

Register file with special treatment of R0 and R15

- reg[0] always outputs 0
- reg[15] is always output, and can be loaded independently from other registers

Effect on state (from programmer's perspective)

- if ld then reg[d] := x
- if ~(ld & d=15) & ldcc then reg[15] := xcc

State update (from perspective of circuit)

- reg[0] there is no state
- reg[d] for $0 < d < 15$: reg[d] := if ld then x else reg[d]
- reg[15] :=

```
if ld & d=15 then x
else if ldcc then xcc
else reg[15]

Inputs
  ld     load control
  ldcc   load into R15
  x      data input
  xcc    R15 data input
  d      destination address
  sa     source a address
  sb     source b address

Outputs
  a = reg[sa]
  b = reg[sb]
  cc = reg[15]
```

4.4 Implementing the register file with special cases

Recursion is controlled by the addresses; if their lengths vary there will be a pattern match error

RFspan determines how to generate the circuit for the base cases

```
data RFspan
  = RFfull      -- contains R0
  | RFhead       -- contains R15
  | RFtail       -- contains R15
  | RFinside     -- contains neither R0 nor R15

headType, tailType :: RFspan -> RFspan
headType RFfull   = RFhead
headType RFinside = RFinside
headType RFhead   = RFhead
headType RFtail   = RFinside

tailType RFfull   = RFtail
tailType RFinside = RFinside
tailType RFhead   = RFinside
tailType RFtail   = RFtail
```

– 1-bit Register file with special cases for lowest and highest – indices (R0, cc). Sigma16 uses 4-bit addresses for d, sa, sb, – leading to 16 registers, where the special cases are R0 (the head, – i.e. lowest address) and R15 (the tail, i.e. highest address)

```
regFileSpec1
  :: CBit a
  => RFspan
  -> a          -- ld: if ld then reg[d] := x
  -> a          -- ldcc: if ldcc then reg[15] := xcc (but ld R15 takes precedence)
  -> [a]        -- d: destination address
  -> [a]        -- sa: source a address
  -> [a]        -- sb: source b address
  -> a          -- x = data input for reg[d]
  -> a          -- xcc = data input for condition code R15
  -> (a,a,a)    -- (reg[sa], reg[sb], reg[15])
```

– Recursion is based on the address words. There will be a pattern – match error if the addresses (d, sa, sb) don't all have the same – number of bits. There will also be a pattern match error if the – base case has RFtype = RFfull, as a singleton register cannot be – both R0 and R15.

– Base cases

```
regFileSpec1 RFinside ld ldcc [] [] [] x xcc = (r,r,zero)
  where r = reg1 ld x
regFileSpec1 RFhead ld ldcc [] [] [] x xcc = (zero,zero,zero)
regFileSpec1 RFtail ld ldcc [] [] [] x xcc = (r,r,r)
  where r = reg1 (or2 ld ldcc) (mux1 ld xcc x)
```

– Recursion case

```
regFileSpec1 rft ld ldcc (d:ds) (sa:sas) (sb:sbs) x xcc = (a,b,cc)
  where (a0,b0,cc0) = regFileSpec1 (headType rft) ld0 ldcc ds sas sbs x xcc
        (a1,b1,cc1) = regFileSpec1 (tailType rft) ld1 ldcc ds sas sbs x xcc
        (ld0,ld1) = demux1 d ld
        a = mux1 sa a0 a1
        b = mux1 sb b0 b1
        cc = cc1
```

– n-bit register file with special cases for R0 and R15

```
regFileSpec
  :: CBit a
  => Int             -- word size
  -> a               -- ld: if ld then reg[d] := x
  -> a               -- ldcc: load R15
  -> [a]             -- d: destination address
  -> [a]             -- sa: source a address
  -> [a]             -- sb: source b address
  -> [a]             -- x = data input for reg[d]
  -> [a]             -- xcc = data input for condition code R15
  -> ([a],[a],[a])  -- (reg[sa], reg[sb], reg[15])

regFileSpec n ld ldcc d sa sb x xcc =
  unbitslice3 [regFileSpec1 RFfull ld ldcc d sa sb (x!!i) (xcc!!i)
                 | i <- [0 .. n - 1]]
```

5 Control

5.1 Control state: basic delay elements

```
st_a = dff ...
st_b = dff st_a
st_c = dff st_b
st_d = dff st_c
...

st_dispatch = dff ...
ps = demux4 op st_a

st_a0 = dff (ps!!0)  -- op=0000 indicates operation a
st_a1 = dff st_a0
st_a2 = dff st_a1

st_b0 = dff (ps!!1)  -- op=0001 indicates operation b
st_b1 = dff st_a0
st_b2 = dff st_a1

st_c0 = dff (ps!!2)  -- op=0010 indicates operation b
st_c1 = dff st_c0
st_c2 = dff st_c1
```

```
...

st_z0 = dff (ps!!15)  -- op=1111 indicates operation z
st_z1 = dff st_z0
st_z2 = dff st_z1
...
```

5.2 Basic delay elemeent method for control

There are many ways to synthesize a control circuit from a control algorithm. A simple approach is the delay element method.

For example, consider the chain of states for the load instruction. In the basic delay element method (which doesn't provide for DMA cycle stealing), the states would be defined like this, and The control signals are generated directly from those states:

```
dff_load0 = dff (pRX!!1)
dff_load1 = dff st_load0
dff_load2 = dff st_load1
```

The control signals are generated by the states. For example, suppose

- State `dff_load0` asserts `c1` and `c2`
- State `load1` asserts `c3`
- State `load2` asserts `1c` and `c3`

Then the controls are defined by

- =c1 = orw [dff=load0, $\text{dff}_{\text{load2}}$]
- =c2 = orw [$\text{dff}_{\text{load1}}$]
- =c3 = orw [$\text{dff}_{\text{load0}}$, $\text{dff}_{\text{load2}}$]

5.3 Enhanced delay elements for DMA and cycle stealing

Direct memory access (DMA) is a method for supporting Input/Output. An input operation requires data from an input device to be stored into the memory. An output operation is the reverse: data must be fetched from memory and sent to an output device.

One way to implement I/O is to require the CPU to perform the memory accesses during an I/O operation. This method was actually used on some very early computers (1940s), but it is extremely slow, and is not used on modern computers.

DMA is far more efficient. The idea is that the processor doesn't access the memory for I/O. Instead, it makes a request for input or output; this means simply sending a small message to the I/O system. The I/O system then performs its own accesses to the memory.

For example "print 80 characters in memory starting at address 2bc3". Sigma16 makes this request using a trap instruction: trap R1,R2,R3 specifies the operation by a number in R1, and arguments in R2 and R3. For example, if R1 contains 1 (the trap code for write), thhis tells the I/O system to print the contents of memory starting at the address in R2, and the number of characters is given in R3.

The main technical issue in DMA is that both the processor and the I/O system are making accesses to the memory, and these are likely to happen at the same time. The memory itself, however, can do only one operation at a time. Therefore it is necessary to ensure that the processor and I/O do not interfere with each other.

Suppose the I/O controller needs to fetch a memory location x. To do so, the system needs to set some control signals, and place x on the memory address control. But these actions could interfere with normal execution of the processor. If the processor happens to be accessing memory at some other address, there will be a conflict.

How can we resolve a confliict between the I/O system and the processor when both want to access memory at the same time? There are two general approaches: cycle stealing and a separate memory management unit. The M1 system uses cycle stealing.

The idea behind cycle stealing is that during every clock cycle, either the processor or the memory is performing an action, but never both. In this context, "action" means changing the state by putting new values into the flip flops.

The main system controller provides a signal DMA that indicates whether the processor or the I/O can perform a memory operation during the current cycle. If DMA is 0 the processor should operate normally. If DMA is 1 the I/O system can access the memory, and the CPU should leave its state unchanged at the next clock tick.

In the basic delay element method, the system will definitly set all the control signals corresponding to the current state. However, we need to

- Set all the processor's control signals to 0 during a cycle when DMA=1.

- Leave the control state unchanged at the next clock tick. That enables the processor to retry its current operation in the next clock cycle.

To achieve this, two signals are defined for every state: a flip flop which represents "the processor is trying to be in this statee, unless the cycle has been stolen", and a logic signal that means "the processor is in charge this cycle and is actually generating control signals".

This To support cycle stealing, there is a dff for each state (e.g. `dff_load0`) and an additional signal (`st_load0`) that is 1 if the flip flip is 1 and the cycle is not stolen. If `dff_load0` is 1 it means that the processor needs to execute this state. If `st_load0` is 1 it means the processor actually is in this state and its control signals can be asserted.

```
dff_load0 = dff (or2 (pRX!!1) (and2 dff_load0 io_DMA))
st_load0  = and2 dff_load0 cpu
dff_load1 = dff (or2 st_load0 (and2 dff_load1 io_DMA))
st_load1  = and2 cpu dff_load1
dff_load2 = dff (or2 st_load1 (and2 dff_load2 io_DMA))
st_load2  = and2 cpu dff_load2
```

Suppose `io_DNA` is 0 for a number of clock cycles, and the machine is dispatching an instruction with secondary opcide 1. That indicates a `load` instruction, and . Consider what happens during a sequence of clock cycles.

- Suppose that during cycle 100 `pRX!!1` is 1.
- Cycle 101
  - At the clock tick beginning the cycle, `dff_load0` will become 1
  - Suppose that during cycle 101, `io_DMA` is 1, so `cpu` is 1. Since `dff_load0` is 1, `st_load0` is also 1, and the control signals for the `load0` state will all be 1. The processor will perform the first step of the `load` instruction. What that really means is that any flip flop changes required will occur st the clock tick that ends Cycle 101 and begins cycle 102.
- Cycle 102
  - At the clock tick between cycles 101 and 102, `dff_load1` becomes 1 and `dff_load0` becomes 0.
  - But suppose that the I/O system needs to steal the cycle to do its own memory access. To do this, the system sets `io_DNA` $= 1$m and that makes `cpu` $= 0$.

  - Although `dff_load1` $= 1$, the corresponding signal `st_load1` is 0 during this cycle. This is because `st_load1 = and2 cpu dff_load1` and the value of `cpu` is 0. Consequently, the control signals set byy `st_load1` all remain 0. Any register updates belonging to `st=load1` will not happen at the next clock tick. Notice that all the cominbational logic calculations for `st_load1` will go ahead; but their results will not be latched into any register at the clock tick. These logic calculations will be repeated during the next clock cycle.
  - Because the processor will not update any registers at the end of the cycle, it is safe for the I/O to set the controls for the memory that it needs, as well as the memory address and data words.
  - At the clock tick ending this cycle, the I/O memory access takes place and the processor does nothing: its cycle has been stolen.

-

6 System