Circuits for Computer Systems

John T. O'Donnell

September 17, 2023

Contents

1 Introduction

This document shows how to use the collection of examples in `CompSysCode/Circuits`. How those circuits work, how they are designed, and how their simulation drivers are designed, are described elsewhere. Here, the focus is on how to execute the circuits using the Hydra hardware description language.

Some of the circuits are small and simple, and there is also a full processor circuit that can execute the programs in the `CompSysCode/Programs/Sigma16/Core` directory.

The circuit simulations require ghc and Hydra. For installation, see `CompSysCode/README-CompSysCode.html`.

Test the installation by entering these commands, which will run a simulation of a binary multiplier circuit:

```
cd CompSysCode/Circuits
ghc -e main Arithmetic/MultiplyRun
```

2 Basic circuits

These small circuits are simple examples illustrating how to specify circuits and simulation drivers. They provide building blocks that are used repeatedly in larger designs. For a circuit named `Xyz`, its simulation driver is named `XyzRun`.

- In a *batch simulation* the driver contains the input test data. This is convenient for repeatable testing.
- In an *interactive simulation* the driver prompts the user for the values of the input signals on each clock cycle.

2.1 Examples

- `SimpleCirc.hs` is a small and simple circuit with just a few logic gates.
  - `ghc -e main Examples/SimpleCircRun`
    - Runs a batch simulation, using inputs defined in the driver (`SimpleCircRun`)
  - `ghc -e main Examples/SimpleCircRunInteractive`
    - Run an interactive simulation that prompts the user for input values every clock cycle.

- `SimDriverRun` illustrates some of the features and formatting capabilities of a simulation driver.
  - `ghc -e main Examples/SimDriverRun`
- `Several` shows how a simulation driver can run several simulations in one driver file.
  - `ghc -e main Examples/Several`

2.2 Mux

- `mux1` is a multiplexer with 1 bit control and two input bits. `mux1 c x y` outputs `x` if `c` is 0; otherwise it outputs `y`. In other words, the output is *(if c=0 then x else y)*
  - `ghc -e main Mux/Mux1Run`
    - Runs a batch simulation
- `mux1w` is a multiplexer with 1 bit control, two input words. Suppose `c` is an input bit, and `w` and `x` are n bit words (for any $n \geq 0$). Then `mux2w c w x` outputs `w` if `c=0`, and `x` if `c=1`. This circuit is useful when you need to select one of two input words.
  - `ghc -e main Mux/Mux1wRun`
  - `ghc -e main Mux/Mux1wRunFormat`
    - Uses automatic formatting in the simulation driver
- `mux2w` has 2 bit control and four input words. Suppose `c` is a pair of bits, and `w`, `x`, `y`, `z` are all n bit words (for any $n \geq 0$). Then `mux2w c w x y z` outputs `w` if `c=00`, `x` if `c=01`, `y` if `c=10`, and `z` if `c=11`. This circuit is useful when you need to select one of four input words.
  - `ghc -e main Mux/Mux2wRun`

2.3 Arithmetic

- `halfAdd` adds two bits and outputs a carry and sum. `halfAdd x y` outputs `(c,s)` where `c` is the carry and `s` is the sum.
  - `ghc -e main Arithmetic/HalfAddRun`
- `add4` 4-bit binary word adder. It takes a carry input bit `c` and two 4-bit words `x` and `y`. `add4 c x y` outputs `(co, s)` where `co` is the carry output bit, and `s` is the 4-bit sum.
  - `ghc -e main Arithmetic/Add4Run`

- `multiply` is a sequential binary multiplier.
  - `ghc -e main Arithmetic/MultiplyRun`

2.4 Register

- `reg1` is a register with 1-bit state. `reg1 ld x` outputs its state, and if `ld` is 1 it puts `x` into the state at the next clock tick.
  - `ghc -e main Register/Reg1Run`
    - Batch simulation
  - `ghc -e main Register/Reg1RunInteractive`
    - Interactive simulation
- `reg` is an n bit register. `reg n ld w` contains a state of n bits, receives an n bit input word `w`. It outputs its state (which is an n bit word), and if the `ld` input bit is 1 it puts `w` into the state at the next clock tick.
  - `ghc -e main Register/RegRun`
- `count4` is a 4-bit binary counter. It is a register whose state is an n bit word. It takes a `reset` input bit, and contains a 4-bit state which is initially 0. At each clock tick the counter increments its state, but if `reset=1` it clears the state back to 0.
  - `ghc -e main Register/Count4Run`
- `bsr4` is a 4-bit bidirectional shift register. It takes an opcode which is a pair of bits that specifies what to do: 0 = no operation; 1 = load x; 2 = shift right; 3 = shift left. It takes a left input `li` and a right input `ri`, and a 4-bit word `x`. The circuit outputs a left output, right output, and its state, and at the next clock tick it updates its state according to the opcode. This circuit illustrates several important circuit design techniques.
  - `ghc -e main Register/BSR4Run`

3 Adding new programs and circuits

It is recommended that you put any new circuits that you define into `CompSysCode/Circuits/Use`
If you have a lot of them, create a new subdirectory, something like `CompSysCode/Circuits/MoreU`
(although with a more descriptive name). This will help keep your circuits
separate from the ones provided in the `CompSysCode` installation.

To make a modified version of an existing circuit, make a complete copy of the existing directory with a new name, and edit the files in the new directory.

For example, suppose you want to modify the `M1` processor circuit to add a new instruction. Make a complete copy of the `M1` directory and name it `M1myversion` (use the command `cp -r M1 M1myversion`). Leave the original `M1` directory unchanged, and make your changes in `M1myversion`.

Go through each of the files in `M1myversion`, and replace `M1` by `M1myversion` in each of the module and import statements (these always appear at the beginning of the file). It's important to do this; if you don't make these changes you will import the original `M1` code instead of your modified `M1myversion code`. For example:

- In `Control.hs`, change `module M1.Control` to `module M1myversion.Control`
- Also `Control.hs`, change `import M1.Interface` to `import M1myversion.Interface`

Make these changes in each of the files in `Circuits/M1myversion`.

Similarly, it is recommended that you put any new programs into `CompSysCode/Programs/Sigma16/Core`. Alternatively, create a new subdirectory `CompSysCode/Programs/Sigma16/Core/MoreUserPrograms`.

4 Register transfer machine

The RTM circuit contains a small array of registers, an adder, and the ability to perform operations like `R1 : 123=` and `R2 : R1 + R3=`. It is like a tiny processor, with only two instructions. However, the RTM doesn't fetch instructions from memory and decode them; its behavior is determined entirely by its control signal inputs. The following will run a simulation of the RTM, using sample data contained in the simulation driver.

```
ghc -e main RTM/RTMrun
```

5 M1 processor circuit

M1 is a digital circuit which is a processor for the Sigma16 Core architecture. It can execute all the example programs in `CompSysCode/Programs/Sigma16/Core`.

The subsystems are defined in separate files, including the ALU, Datapath, and Control. The signals that communicate between the subsystems are defined in Interface. The processor and memory are defined in System. The simulation driver is defined in M1run.

There is also a utility program `ReadSigma16Obj.hs` that reads an object code file, parses it, and generates the inputs needed to load the code into the M1 memory. You don't need to do anything with this file.

The simulation driver writes two log files as the processor runs. These log files can be deleted any time, and an old log file is overwritten when a new simulation runs.

- `logCircuit.txt` contains the simulation driver output, showing the internal signals and registers for each clock cycle.
- `logWrite.txt` contains text that is output by the Sigma16 program, if any.

5.1 Quick start: running a program on the circuit

Here is a simple example program `Add`, with comments removed. The assembly language source is in the file `CompSysCode/Programs/Sigma16/Core/Simple/Add.asm.txt`. There is also an assembly listing file `Add.lst.txt` and an object code file `Add.obj.txt`.

```
   7 0000 f101 0008       load   R1,x[R0]   ; R1 := x
   8 0002 f201 0009       load   R2,y[R0]   ; R2 := y
   9 0004 0312            add    R3,R1,R2   ; R3 := x + y
  10 0005 f302 000a       store  R3,z[R0]   ; z := x + y
  11 0007 c000            trap   R0,R0,R0   ; terminate
...
  17 0008 0017      x     data  23
  18 0009 000e      y     data  14
  19 000a 0000      z     data   0
```

It's a good idea first to run the program on the Sigma16 app. Go to the Sigma16 Home page at `https://jtod.github.io/home/Sigma16/` and click on the link to launch the app. Go to the `Examples` tab, select `Core instruction set`, then `Simple`, then `Add.asm.txt`. Go to the `Assembler` tab and click `Assemble`, then the `Processor` tab and click `Boot`. This will read the object code into memory, and you can step through the execution of the program (click `Step`), or run it to completion (click `Run`). See the User Guide for documentation of the instruction set and more about how to run programs.

To run the program on the circuit, enter the following commands. Lines beginning with `$` are shell commands, and lines beginning with `M1>` are commands to the simulation driver `M1run`. (Don't enter the prompts `$` or `M1>`.) The meanings of the commands are explained below.

```
$ ghc -e main M1/M1run
M1> boot Simple/Add
M1> run
M1> regs
```

```
M1> mem 0 10
M1> quit
$
```

- `ghc -e main M1/M1run` tells the shell to run the simulation driver.
- `boot Simple/Add` When the circuit starts, the memory contains all 0. This command tells the driver to read the machine language program (`Add.obj.txt`) and store each word of object code into memory. This process is called *booting*.
- `run` The command starts the control algorithm, which repeatedly fetches the next instruction and executes it. When the program finishes it executes a `trap R0,R0,R0` instruction to stop. The driver detects this and pauses the simulation.
- `regs` Now we tell the driver to print out the contents of all the registers. The program leaves the result, which is 37 (hex 0025), in `R3`, and you can check that R3 has the right value.
- `mem 0 10` This prints the contents of memory from address 0000 to 000a. (In the command, addresses are specified in decimal.) The program stores the final result into the variable `z`. To find it, we need to know the address of the variable. Check the assembly listing, `Add.lst.txt`, and you can see that the address of `z` is 000a. The mem command shows that the contents of memory at that address holds the correct result.
- `quit` Terminate the simulation driver and return to shell.

5.2 ALU

You can run the ALU on its own with suitable test data. This makes it easier to test the ALU than running it as a part of the full processor circuit.

```
ghc -e main M1/ALUrun
```

5.3 M1 System

5.4 Object code file

Before the processor can execute a program, that program must be present in the memory. This is achieved by the `boot` command: the process of reading an object code file and writing it into memory is called *booting* (short for *bootstrapping*).

5.4.1 Translate assembly language to object code

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

- *Sigma16 app.* To launch Sigma16, visit `https://jtod.github.io/home/Sigma16/`. Load a program (or type it in to the editor), assemble it, and save the object code in a file. To see the object code, click the Object Code link in the Assembler page. Then copy and paste the text into a text editor and save it. If the source file is `Add.asm.txt`, the object file must be named `Add.obj.txt`

- *Command.* If you have installed the Sigma16 tools on your computer, go to the `Programs/Sigma16/Core/Simple` directory and enter `sigma16 assemble Add`.

- *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. Hand assembly is particularly useful when experimenting with new instructions in an architecture.

5.4.2 File naming convention

For an assembly language module named x, there are several files distinguished by the file extension.

- `x.asm.txt` is the assembly language source file, written by a programmer (or a compiler).

- `x.obj.txt` is the object code file, which contains machine language code.

- `x.lst.txt` is the assembly listing: this shows the source program, the object code for each instruction, the symbol table, and any error messages.

- `x.md.txt` is metadata used by the emulator; the programmer doesn't need to do anything with this file.

An assembly language source program has a name like `Simple/Add.asm.txt` and the corresponding object code (machine language) file has a name like `Simple/Add.obj.txt`. The `boot` command uses just the base name of the file, for example `boot Simple/Add`.

5.4.3 Finding the object file

When the M1 simulation driver executes a `boot` command, it looks for the object code file using a search path. The default is to look for the object code in the `CompSysCode/Programs/Sigma16/Core/` directory. For example, if you enter the command `boot Simple/Add`, the driver will look for `CompSysCode/Programs/Sigma16/Core/Simple/Add.obj.txt`.

You may wish to keep the Circuits directory and your Sigma16 programs directory in different locations in your file system. In this case, it may be convenient to override the default and tell the system to look for the object code somewhere outside `CompSysCode`. To do so, make a file named `Circuits/M1/progdir.txt`, and put the full file path of the directory containing your programs in the first line of that file.

5.5 Running the M1 processor

You need to be in the `CompSysCode/Circuits` directory. The following commands will start M1, boot an example program, and run it to completion. The object code must be in Sigma16 object format, which is produced by the Sigma16 assembler. Assembling source program `Add.asm.txt` will produce the machine language code in `Add.obj.txt`.

```
ghc -e main M1run
M1> boot Simple/Add
M1> run
M1> quit
```

The first command (`ghc -e main M1run`) launches the M1 simulation driver. When the circuit starts, its memory is all 0. To run a machine language program we need to read in the object code file (Simple/Add.obj.txt). The `boot` command tells the simulation driver to read the object file and generate the control signals needed to write the code into memory, starting from address 0.

The run command tells the driver to execute clock cycles repeatedly, until either a breakpoint is encountered or the program halts.

Alternatively, you can run one clock cycle simply by pressing Enter instead of typing run. Do this repeatedly to see how the circuit evolves through a sequence of clock cycles.

If the object code is n words long, it will take the circuit n clock cycles to boot it. You can reach the point when the circuit actually starts running the program by entering these commands. The `break reset` command says that `run` should stop execution when the `reset` signal becomes 1.

```
$ ghc -e main M1run
M1> boot Simple/Add
```

```
M1> break reset
M1> run
M1> quit
```

Now you can step through the execution of the program: pressing Enter will run just one clock cycle.

To run the program to completion (unless a breakpoint is set), enter `run`. To exit from the simulation driver and get back to the shell, enter `quit` or `q`.

Here is a brief example of using breakpoints and register/memory display. The ArrayMax program finds the largest element of an array of constant data. The correct result is 40 decimal, which is 0028 hex. The program leaves the result in R4, and also stores it into `max` whose address is 0018. By entering the `regs` command you should see that R4=0028 and by entering `mem 0 30` you should see that mem[0018]=0028.

```
$ ghc -e main M1run
M1> boot Arrays/ArrayMax
M1> break reset    set breakpoint
M1> run            run until reset=1
M1> (enter)        run just one  clock cycle
M1 run
M1> regs           print the contents of the register file
M1> mem 0 30       print memory from address 0 to 30
```

5.6 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
; Sigma16  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
```

5.7 Reading the simulation output

The driver reads the object code file and siaplays the object code, which is a list of numbers. Then it generates the inputs to the circuit that will be required to boot the program. This is a list of strings; each string gives the input signal values for one clock cycle. These signals tell the circuit to perform an I/O operation and that the operation is an external read into memory. Furthermore, the memory address to use and the data value are specified. For example, the first element of the list gives the inputs for clock cycle 0, and the last two numbers mean that the input will store into address 0 and the value to store is 61697.

```
Object code is [61697,8,61953,9,786,62210,10,45056,23,14,0]
Boot system inputs = ["0 1 1 0 0 0 61697","0 1 1 0 0 1 8",
"0 1 1 0 0 2 61953","0 1 1 0 0 3 9","0 1 1 0 0 4 786",
"0 1 1 0 0 5 62210","0 1 1 0 0 6 10","0 1 1 0 0 7 45056",
"0 1 1 0 0 8 23","0 1 1 0 0 9 14","0 1 1 0 0 10 0"]
M1>
```

The purpose of the simulation driver is to show the most important of the circuit signals, so it produces a lot of output as it runs. Here is a portion of the output.

The whole output

5.8 M1 simulation driver commands

The simulation driver, M1run, interacts with the user through a command line interface. When it gives a prompt `M1>` it waits for the user to enter a command. This section describes the M1 driver commands.

5.8.1 help

The `help` command prints a list of the commands.

5.8.2 quit

The `quit` command terminates the driver.

5.8.3 boot

The `boot` command has one argument, a filename. It reads the file, which must be a valid Sigma16 object code file, and generates the control signals needed to store the code into the memory. The filename is appended to the string in `fileprefix.txt`, if that file exists. This lets you set up a default directory for object code files.

Suppose `fileprefix.txt` contains `Programs/Core`. Then the following command will boot the object program `Programs/Core/Simple/Add.obj.txt`.

```
boot Simple/Add
```

The `boot` command works by setting the circuit inputs to perform input operations in order to store the object code. It takes one clock cycle for each word. Therefore when you enter a `boot` command, the memory won't change immediately; instead, each clock cycle causes one more word of object code to be stored. You can see the effect by issueing a `boot` command, and then stepping one clock cycle at a time. Use the command `mem 0 10` to display the first few words of memory.

Normally you won't want to watch the circuit signals for each clock cycle during a `boot`. You can set a breakpoint to stop execution when the boot has finished, and then use `run` to simulate enough clock cycles at full speed to complete the boot.

```
boot Simple/Add
break reset
run
```

5.8.4 cycle (or just enter an empty command)

The `cycle` command executes one clock cycle on the circuit. At the end of the cycle the driver displays some of the signals in the circuit.

Just pressing enter at a driver prompt is equivalent to entering `cycle`. The usual way to step through a sequence of cycles is to press enter several times.

5.8.5 regs

The simulation driver shows the values of all the output signals from the circuit, and this includes key registers, such as `pc`, `ir`, and `adr`. However, most of the computer's state is in the register file and the memory, and these are not directly visible.

If you follow all the details of every clock cycle, you can work out the contents of the register file and the memory. But this may be impractical. If you want to know what is in memory at some particular address, there is

no bound on how far back you would have to search to find the point when something was stored in that location.

The `regs` command displays the contents of the register file. It doesn't take an argument. This command generates the control signals required to make the processor circuit read out the registers, one per clock cycle. The command gathers the results and prints out the registers. It requires one clock cycle for each register, and uses the dma facility to read out the registers without disturbing an executing program.

```
regs
```

The `regs` and `mem` commands are not implemented by looking into the simulator's internal data structures. Indeed, the simulator doesn't know anything about the circuit apart from the signal values. The commands are implemented by the Input/Output system, using direct memory access (DMA) and cycle stealing. This is the way testing is done on real hardware. You can see that the dump commands require a number of clock cycles to perform, even though the driver doesn't show all the internal signals during those cycles.

5.8.6 mem a b

The `mem a b` command prints the contents of memory from address a through address b. The arguments a and b are specified as decimal integers. This command will print memory from address 0 to 30 decimal:

```
M1> mem 0 30
```

The `mem` command, like `regs`, is implemented using DMA and cycle stealing; see the `regs` section.

5.8.7 break x

The machine may execute 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; the name of this signal is the argument `x` to the break command. After setting the breakpoint, you can enter a run command. At the start of each cycle the driver checks the signal, and if it is 1 the driver stops the simulation and gives a prompt. Thus the driver will run the circuit at full speed until the breakpoint signal becomes 1. At that point you can examine the machine

state in detail, look at the registers or memory, step through some cycles, and resume execution with another `run` command.

The break signal `x` cannot be any arbitrary signal; it must be registered in advance. The `help` command gives you a list of signals that are registered so they can be used as a breakpoint.

One useful breakpoint signal is `reset`. When you start the system it may take a considerable number of clock cycles to boot the machine language program. You can skip over those cycles and go directly to the point where the machine starts executing the program with the following commands. 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.

```
break reset
run
```

Another useful technique is to go quickly to the point where the machine is starting to execuate a particular instruction that you're interested in. The convention is that the first state of the control algorithm for an instruction is named `st_instr0`. For example, if you want to watch in detail how the circuit executes a load instruction, use these commands:

```
break st_load0
run
```

5.8.8 Writing simulation output to a file

You can tell the simulation driver to copy the simulation output to a log file by entering this command:

```
logging on
```

The output will be saved in `Circuits/logCircuit.txt`. The command to turn logging back off is

```
logging off
```

The default is `logging on`.

5.8.9 Showing signals during DMA cycles

Input/Output takes place during

- A trap read (when the program running on M1 requests input)
- A trap write, when the program requests output.

- A regs command. This tells the simulation driver to perform DMA operations to fetch and print the contents of the register file.
- A mem command. This tells the driver to perform DMA operations to fetch and display a block of words from memory.

M1 performs input/output using direct memory access (DMA) and cycle stealing. During I/O the normal display of all the signals for each clock cycle is suppressed, and only the results of the actual DMA operations are displayed. If you want to examine all the signals, for example in order to understand how the DMA works, enter `dmasigs on`. To go back to the more concise output, enter `dmasigs off`.

6 Using ghc or ghci

The standard way to run Hydra circuit simulations is to use the ghc command. The `-e main` switch says to execute the main program, and the last argument is the module to execute, which is the simulation driver.

```
$ ghc -e main M1/M1run
M1> boot Simple/Add
M1> run
M1> quit
```

Instead of ghc, you can use ghci, which is an interactive Haskell interpreter that provides some useful debugging commands. Specify the simulation driver with the `:load M1run` command, and then start the driver with the `:main` command.

```
$ ghci
ghci> :load M1/M1run
ghci> :main
M1> boot Simple/Add
M1> run
M1> quit
ghci> :quit
```

The ghci command `:main` takes an optional argument which is the object file to boot (in the example above, it's `Simple/Add`). This file can be omitted: just enter `:main`. This will start the M1 driver, and at its prompt (`M1>`) you can enter `boot Simple/Add`.

```
$         is the bash shell prompt
ghci>     is the ghci prompt
M1>       is the simulation driver prompt
```

Here are a few useful ghci commands. See the ghc User Guide for full documentation.

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