18-240: Structure and Design of Digital Systems Electrical & Computer ENGINEERING

RISC240 Reference Manual

Bill Nace & Saugata Ghose

Version 1.0.2
22 April 2019

1. Introduction

RISC240 is a special-purpose *instruction set architecture* (ISA) designed for use in the Carnegie Mellon 18-240 class. The RISC240 ISA consists of word size and addressing definitions (Section 3), the visible state (Section 4), and assembly instructions (Section 5). You can use the assembly instructions and assembler *pseudo-operations* (Section 6) to write assembly programs (Section 7). We provide the as240 assembler (Section 8) and the sim240 simulator (Section 9) to assemble and test assembly programs written using the RISC240 instructions, and we provide synthesizable SystemVerilog for a working RISC240 CPU (Section 10).

RISC240 is based upon the RISC-V ISA (`https://riscv.org/`), but has been ***significantly modified and simplified*** for educational purposes. As a result, a number of key components were not included in the design of RISC240, to ease the learning curve for introductory students. RISC240 is not intended to be a high-performance ISA, so don't expect superscalar features, pipelining, multi-threading, etc. You will learn plenty about those features, and how a real-world ISA such as RISC-V enables them, in your computer architecture class (e.g., 18-447 at CMU).

2. Overview on ISAs

An instruction set architecture is an interface specification – a set of guidelines – about how to program a CPU or family of CPUs. Sometimes we say something like that "an ISA is a *contract* between the chip designer (i.e., the computer architect) and the programmers." What this means is that both sides (the programmer and the chip designer) need to have a *common vision* about how to get the CPU to run programs properly. The chip designers (typically a team of designers) have built the CPU to work in a certain way, and they need to be able to tell programmers these details. However, the chip designers don't just want to give programmers a circuit diagram or a datapath picture — most programmers aren't computer architects, so those communication methods don't work well. Instead, the ISA is a list of instructions and other details about how the chip works, without exposing the *complete* specification of how the chip works.

If the complete specifications of a chip were exposed, programmers may write software that depends on specific details about this chip. This would prevent designers from changing those specifications on future chips, which could severely limit improvements to chip performance. By exposing *only enough* information about the interface between programs and the chip, chip designers can build new chips with significant under-the-hood changes, and programmers can run their existing programs on these new chips without having to rewrite or recompile their code.

3. RISC240 Word Sizes and Addressing

The RISC240 ISA makes use of ***16-bit words***. In other words, all operations are done on 16 bits of data at a time. Recall that a byte consists of 8 bits, which means that one word in RISC240 is made up of two bytes.

RISC240 assumes that all addresses are ***byte-addressable***. This means that each byte in memory has its own address. Because everything in the RISC240 operates at the granularity of a *word* and *not* a byte, we simplify the ISA by providing *automatic word alignment*: the least-significant bit of an address is *always* replaced by a zero when accessing memory (and in special registers such as the program counter; see Section 4), ensuring that no operation takes place on multiple words at once. For example, if you try to branch to address `0x0053`, a RISC240 CPU will automatically set the program counter to `0x0052`.

4. Visible State in RISC240

The assembly language programmer selects instructions and orders them such that they have a particular effect on a RISC240 CPU. Through this process, the programmer is manipulating the visible state of the CPU.[1] The RISC240 ISA provides the following pieces of visible state:

- **Registers (r0 – r7):** Eight general-purpose registers, each 16 bits wide. The registers are named r0 through r7. Register r0 is known as the *zero register* – it is hardwired to the value zero, and any writes to r0 are ignored by the CPU.

- **Program Counter (PC):** A 16-bit wide register that, at the beginning of the first clock cycle of the execution of an instruction, holds the memory address of that instruction.[2]

- **Condition Code Flags (CC *or* ZCNV):** Four single-bit flags that are set based on the outcomes of an ALU operation. We usually list them in the order **`ZCNV`**, where each of the letters represents one of the following flags:
  - ***Zero Flag*** **(`Z`):** This bit is set (i.e., **`Z`**=1) if the result of the ALU operation was zero.
  - ***Carry Flag*** **(`C`):** This bit is the carry-out bit of the ALU operation.
  - ***Negative Flag*** **(`N`):** This bit is the most significant bit of the result of the ALU operation. Therefore, if it is set (i.e., **`N`**=1), that means the result was a negative value.
  - ***Overflow Flag*** **(`V`):** This bit is set if the ALU operation resulted in overflow.

- **Memory:** RISC240 includes a specification of memory, unlike many CPUs. In our case, the memory is an array of 65,536 (i.e., 2^{16}) *bytes*, where reads and writes are done on 16-bit *words*. As such, it has a 16-bit address and a 16-bit data bus. **`M[8]`** represents the 16-bit word stored in byte addresses 8 and 9. The memory has combinational read and sequential write capabilities.

[1] The programmer is also manipulating non-visible state, but they don't need to know about those bits – this is the entire point of an ISA.

[2] The reason that this definition is so tortured is that the PC is changed at different points during the execution of different instructions. At various instances, it might hold the location of the second word in a two-word instruction, the instruction following the current one in memory, or the instruction to be executed next (a branch target, for instance).

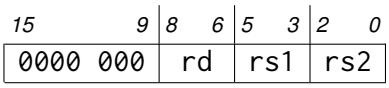
5. RISC240 Assembly Instructions

ADD rd, rs1, rs2

Addition

Semantics: `rd ← rs1 + rs2`
CC Flags: `ZCNV - set normally`

Encoding:



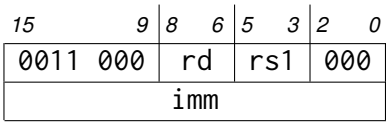
Cycles: 5

ADDI rd, rs1, imm

Add Immediate

Semantics: `rd ← rs1 + imm`
CC Flags: `ZCNV - set normally`

Encoding:



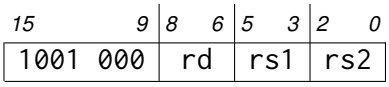
Cycles: 7

AND rd, rs1, rs2

Bitwise AND

Semantics: `rd ← rs1 · rs2`
CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Encoding:



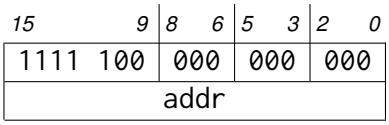
Cycles: 5

BRA addr

Branch Always

Semantics: `PC ← addr`
CC Flags: `not changed`

Encoding:



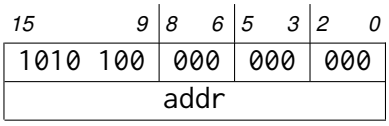
Cycles: 7

BRC addr

Branch If Carry

Semantics: `if(C) PC ← addr`
CC Flags: `not changed`

Encoding:



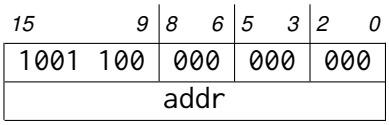
Cycles: 7 if taken; 6 if not taken

BRN addr

Branch If Negative

Semantics: `if(N) PC ← addr`
CC Flags: `not changed`

Encoding:



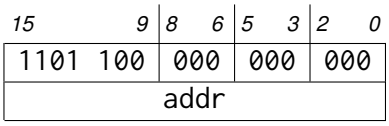
Cycles: 7 if taken; 6 if not taken

BRNZ addr

Branch If Negative or Zero

Semantics: `if(N OR Z) PC ← addr`
CC Flags: `not changed`

Encoding:



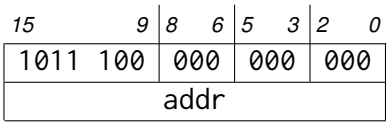
Cycles: 7 if taken; 6 if not taken

BRV addr

Branch If Overflow

Semantics: `if(V) PC ← addr`
CC Flags: `not changed`

Encoding:



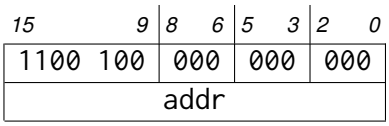
Cycles: 7 if taken; 6 if not taken

BRZ addr

Branch If Zero

Semantics: `if(Z) PC ← addr`
CC Flags: `not changed`

Encoding:



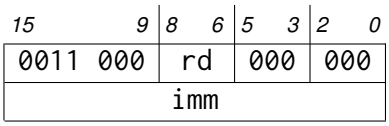
Cycles: 7 if taken; 6 if not taken

LI rd, imm

Load Immediate

Semantics: `rd ← imm`
CC Flags: `ZCNV - set normally`

Encoding:



Cycles: 7

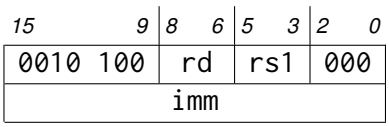
Notes: implemented in hardware as ADDI rd, r0, imm

LW rd, rs1, imm

Load Word

Semantics: `rd ← M[rs1 + imm]`
CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Encoding:



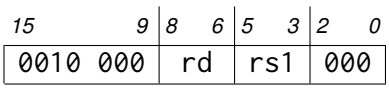
Cycles: 9

MV rd, rs1

Move Register

Semantics: `rd ← rs1`
CC Flags: `not changed`

Encoding:

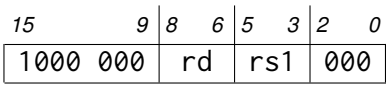


Cycles: 5

NOT rd, rs1

Bitwise NOT

Encoding:



Semantics: `rd ← NOT rs1`

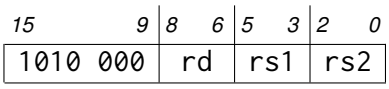
CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 5

OR rd, rs1, rs2

Bitwise OR

Encoding:



Semantics: `rd ← rs1 OR rs2`

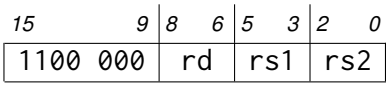
CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 5

SLL rd, rs1, rs2

Shift Left Logical

Encoding:



Semantics: `rd ← rs1 << rs2[3:0]`

CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 5

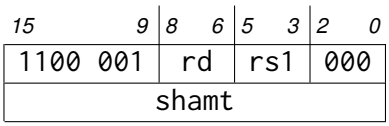
Notes: for a shift amount of *n*, shifts in *n* zeros from the right

---

SLLI rd, rs1, shamt

Shift Left Logical Immediate

Encoding:



Semantics: `rd ← rs1 << shamt[3:0]`

CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 7

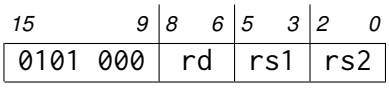
Notes: for a shift amount of *n*, shifts in *n* zeros from the right

---

SLT rd, rs1, rs2

Set If Less Than

Encoding:



Semantics: `if(rs1 < rs2) rd ← 16'b1`
`else rd ← 16'b0`

CC Flags: `ZCNV - set based on result of (rs1 - rs2)`

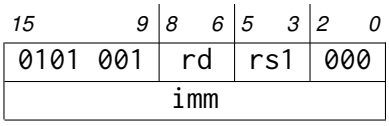
Cycles: 6

Notes: two's complement comparison

SLTI rd, rs1, imm

Set If Less Than Immediate

Encoding:



Semantics: `if(rs1 < imm) rd ← 16'b1`
`else rd ← 16'b0`

CC Flags: `ZCNV - set based on result of (rs1 - imm)`

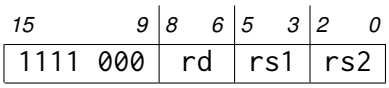
Cycles: 8

Notes: two's complement comparison

SRA rd, rs1, rs2

Shift Right Arithmetic

Encoding:



Semantics: `rd ← rs1 >>> rs2[3:0]`

CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 5

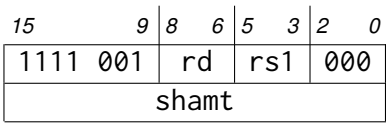
Notes: for a shift amount of *n*, shifts in *n* copies of `rs1[15]` from the left

---

SRAI rd, rs1, shamt

Shift Right Arithmetic Immediate

Encoding:



Semantics: `rd ← rs1 >>> shamt[3:0]`

CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 7

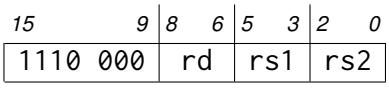
Notes: for a shift amount of *n*, shifts in *n* copies of `rs1[15]` from the left

---

SRL rd, rs1, rs2

Shift Right Logical

Encoding:



Semantics: `rd ← rs1 >> rs2[3:0]`

CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 5

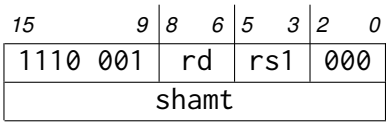
Notes: for a shift amount of *n*, shifts in *n* zeros from the left

---

SRLI rd, rs1, shamt

Shift Right Logical Immediate

Encoding:



Semantics: `rd ← rs1 >> shamt[3:0]`

CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 7

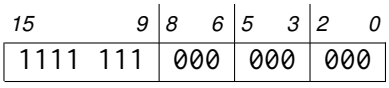
Notes: for a shift amount of *n*, shifts in *n* zeros from the left

---

STOP

Stop Execution

Encoding:



Semantics: `PC ← PC`

CC Flags: `not changed`

Cycles: 5

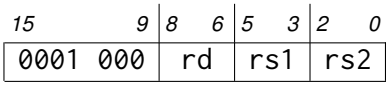
Notes: used to halt the CPU when program execution is finished, by looping infinitely on the STOP instruction until a reset signal is asserted

---

SUB rd, rs1, rs2

Subtraction

Encoding:



Semantics: `rd ← rs1 - rs2`

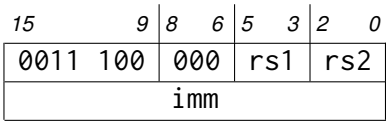
CC Flags: `ZCNV - set normally`

Cycles: 5

SW rs1, rs2, imm

Store Word

Encoding:



Semantics: `M[rs1 + imm] ← rs2`

CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Cycles: 9

XOR rd, rs1, rs2
Bitwise XOR

Semantics: `rd ← rs1 ⊕ rs2`
CC Flags: `ZN - set normally`
`C = 0`
`V = 0`

Encoding:

15 9	8 6	5 3	2 0
1011 000	rd	rs1	rs2

Cycles: 5

---

6. Assembler Pseudo-Operations

Pseudo-operations are directives that show up in an assembly program, but are intended to communicate with the assembler as it prepares the machine language. Pseudo-operations don't communicate anything to the CPU (unlike an instruction), and so are not part of the ISA. Unlike an instruction, there is no machine language equivalent for each pseudo-operations: they don't show up in the machine language file, though they do effect how the assembler shapes the machine language. For our **as240** assembler (which assembles RISC240 assembly programs into machine language), we have three pseudo-operations (all of which start with a period):

- **.ORG addr**
  Origin

  This pseudo-operation tells the assembler that the following instruction should be placed at address **addr**. Normally, when **.ORG** is *not* used, the next line will have an address that is zero, two, or four more than the current line (depending on if this line is blank, has a short instruction on it, or has a long instruction on it). The **.ORG** pseudo-operation interrupts this incremental processing and allows you to put data or instructions at any place in the RISC240 memory map.

- **.DW value**
  Data Word

  This pseudo-operation tells the assembler that you want a particular value to be placed in memory at the address of this line. The value can be specified as a hexadecimal value or as a label.

- **.EQU value**
  Equals

  This pseudo-operation assigns a constant value to a particular label. The label on this line does not equate to an address, unlike all other labels. Rather, the compiler finds all instances where the label is used as an operand, and replaces them with the constant value when converting assembly into machine code.

7. Assembly Program Format

An assembly program is written as a text file, in ASCII format, much like other programming languages. The file is named with a **`.asm`** extension.[3] Assembly programs are line-oriented, which means that the line stands alone. All information about a single instruction is contained in a single line. There is no way to have an instruction span multiple lines. On the other hand, not all lines have an instruction on them — they may be blank, contain a comment or a pseudo-operation. The entire assembly file is case-insensitive. You may use lower or upper case characters, but the assembler looks at the file as if all characters had been converted to upper case.

On any line of the assembly file, up to four fields will occur, in the following order.[4] Any line containing a label or operand must also have an instruction. Each field is separated by some form of whitespace (spaces, tabs, etc.).

1. **Label**

   The label specifies a human-usable name for the address of the instruction on that line. Labels also can be used to name a constant specified in an **`.EQU`** pseudo-operation.

   The label is an alphanumeric value starting at the beginning of the line. There should be *no whitespace before the label*. Labels are *case insensitive* (as is the entire file). Labels may be any length. Each label must be unique — there can be no other label in the file with the same value.

   You may use any combination of letters, digits, and the underscore ('_') character, for a label, though we strongly recommend that it be meaningful. Also, labels that look like register names (**`r4`**) or instruction mnemonics (**`LDA`**) are frowned upon, though legal.

2. **Instruction**

   This is the mnemonic for the instruction. It must match one of the given instructions in the RISC240 ISA (Section 5), or one of the assembler pseudo-operations (Section 6).

3. **Operands**

   Depending upon the instruction, there should be 0–3 operands. If there are two or more operands, they *must* be separated by a comma, and *may* have whitespace before or after the comma.

   The nature of the operand is determined by the instruction. Operands can be register names, immediate values, memory addresses, or labels.

[3] This isn't a hard-and-fast rule. as240 expects it by default, but you may specify any extension. Naming your assembly language files with a .c or .java extension is asking for trouble.

[4] Note that blank lines are legal, as they have zero of the fields.

Register names start with the '**r**' character, and are followed by a single digit ranging between 0 and 7, inclusive.

Immediate values and *memory addresses* can be numerical values or label names. Numerical values are hexadecimal numbers of no more than 16 bits (i.e., 0 through FFFF). All hexadecimal values must start with the dollar sign ('**$**'). Leading zeros are not necessary.

Label names must match a value in the label field somewhere in your assembly code.

4. **Comment**

   A comment is specified by using the semicolon ('**;**'). All text on the line following the semicolon is ignored by the assembler.

If a field does not exist (like a label), the separation whitespace must still exist. Therefore, a label is the only thing that can be at the beginning of a line. A line without a label needs at least one space before the instruction.

8. RISC240 Assembler: as240

as240 is an assembler for the RISC240 ISA. You can find it in the `/afs/ece/class/ece240/bin` directory, and can execute it from there. You are welcome to make a copy of the assembler program so you can execute it from your own machine or in your own AFS space. Note that it is a Python 3 program, and will require a Python 3.5+ installation to execute.

Note that the directory `/afs/ece/class/ece240/bin` is not writable, which means that **you will have to make sure that your current working directory is some other directory** where you have write permissions (your own AFS space, for instance). When working in another directory, you can run the assembler by either (1) specifying the full path when you run as240:

```
> /afs/ece/class/ece240/bin/as240 lab4.asm
```

or (2) including `/afs/ece/class/ece240/bin` in your path, which allows you to call the assembler locally:

```
> as240 lab4.asm
```

When you run the assembler, it takes the following arguments:

```
> as240 [options] <base_name>.asm
```

When specifying the file to assemble (**<base_name>.asm**), as240 assumes a default extension assumed of **.asm**, but this can be overridden by specifying the full filename. as240 produces three files once it finishes running:

- **List File:** Shows your code and how it was converted by the assembler. All of the memory values for all of the instructions are shown, as well as label values, data words, etc. By default, the **`.list`** file has the same base name as the **`.asm`** file (e.g., assembling lab4.asm produces lab4.list)
- **Simulation Memory Image File:** Shows all of the addresses and the values that should be stored in each address. By default, the file is called **`memory.hex`**.
- **Synthesis Memory Image File:** Like the **`memory.hex`** file, shows addresses and values that should be stored in each address. This is a different format file, used by the memory model for synthesis. This file is called **`memory.mif`**. At the current time, there is no way to use a different output filename for this file.

The following command-line options are available for as240:

`-h` **`--help`**	provides short help text and usage information, and exits
`-m <filename>` **`--mfile=<filename>`**	uses the specified filename for the memory image file
`-l <filename>` **`--listfile=<filename>`**	uses the specified filename for the list file
`-s <filename>` **`--symbolfile=<filename>`**	outputs a symbol list at the specified filename
`-o`	sends the list file output to **`stdout`** (for piping into sim240) instead of a file, and writes no files to the directory
`--version`	prints the current version number, and exits

9. RISC240 Simulator: sim240

sim240, the RISC240 simulator, is a great resource to understand (micro-)instruction operations and debug programs. Like as240, sim240 can be found in the `/afs/ece/class/ece240/bin` directory, and is a perl program that can be copied to another machine for your own use.

When you run the simulator, it takes the list file generated by as240 (e.g., lab4.list) as an argument:

```
> sim240 [options] <base_name>.list
```

The following command-line options are available for sim240:

-h **--help**	provides short help text and usage information, and exits
-r **--run**	run only: doesn't accept user input, but rather starts the program, and outputs all simulation steps until a STOP instruction is reached (or some large number of clock cycles has passed)
-n **--norandom**	initializes each word in memory to zeros, instead of to random values
-t <filename> **--transcript=<filename>**	outputs a transcript of the simulator at the specified filename
-q **--quiet**	doesn't print output with step/ustep
-g <filename> **--grade=<filename>**	runs simulation, checks state against the specified file, and exits
-i	reads the list file from **stdin** instead of from a list file (this lets you pipe your as240 output to sim240 without generating **.list**/**.hex** files)
-v **--version**	prints the current version number, and exits

9.1. sim240 Commands

Unlike as240, sim240 is an *interactive* program that allows the user to specify what should happen throughout its use. When the simulator is invoked, a prompt will be printed on the terminal and the simulator will await user input. At any prompt, the user can enter any of the following commands:

quit **q** **exit**	quits the simulator
help **h**	prints a help screen

reset	resets the processor state to where it was when the .list file was loaded: memory and registers (including PC) return to initial values, and all breakpoints are cleared
run [n] **r [n]**	simulates the next *n* instructions; if *n* is not specified, simulate until a STOP instruction is encountered; if a breakpoint is encountered, it will halt the run
step **s**	simulates a single instruction; technically, simulates until the next FETCH state is encountered, so if you **ustep** several states into an instruction, **step** will only complete simulation of the current instruction
ustep **u**	simulates a single micro-instruction; that is, a single state or single clock
break [addr/label]	sets a breakpoint; if the address specified (or the address of the label specified) is encountered during a run, the simulator will stop at that point (this is incredibly useful for getting into the program to the point where you think there might be trouble, rather than stepping 87 times, or checking to see if some code is ever executed)
lsbrk	lists all the breakpoints that you have previously set
clear [addr/label]	clears a breakpoint that you have previously set for this address
save <filename>	saves the current state of the simulation to a file; includes your current location in the program, all memory contents, and all breakpoints
load <filename>	loads a simulation file that was created earlier using the **save** command

9.2. Querying RISC240 State

You may ask about the contents of memory or a register by using the question mark command. Simply specify a register (including microarchitectural registers like MAR and IR) and then follow it with a question mark:

```
> r3?
> MAR?
> IR?
```

If you want to see the entire register file, then you can use the wildcard asterisk:

```
> r*?
r0: 0000      r1: 1492
r2: 0410      r3: 1234
r4: BEEF      r5: 0000
r6: 8787      r7: A492
```

You can also view all microarchitectural registers (e.g., MAR, IR) with the **`*?`** command, though the output isn't quite as pretty:

```
> *?

0208 FETCH 283E 0000 0000 283D 0000 0000 1492 0410 1234 BEEF 0000 8787 A492
```

Similarly, you can ask about a memory location. Use the m array notation with an address between square brackets. You may use **`m`** or **`mem`** (or their capitalized versions). You need not specify leading zeros. You must use **an address in hexadecimal format**, and not a label. By using a colon to delimit start and end addresses, you can ask about a range of values. Recall that memory is byte-addressable (see Section 3), but note that sim240 will always print the entire word in a word-aligned manner automatically.

```
> m[0000]?
> M[1234]?
> mem[20:87]?
```

For each memory address, you will be told the value that the word at that location holds, as well as what that word decodes to if it were an instruction. For example:

```
> m[7:a]
mem[0006:0007]: 31C0 ADDI 7 0 0
mem[0008:0009]: 0001 ADD 0 0 1
mem[000A:000B]: F0F0 SRA 3 2 0
```

Note that this example is asking about *two* instructions: **`ADDI`** and **`SRA`**. **`ADDI`** is a two-word instruction, but the printing routine doesn't try to disassemble the values stored in memory. As a result, the value stored in **`mem[0008]`**, $0001, is printed as a **`ADD`** instruction, even though it is supposed to be an immediate value, because that's what the first word of a **`ADD`** instruction looks like. Just be a bit careful about interpreting what you see.

9.3. Setting RISC240 State

You can use similar means to change the state of a register or memory. In this case, instead of ending with a question mark, use an equals sign:

```
> r2=d2
> MAR=0100
> mem[290]=7734
```

Again, when you change the value in a memory location, sim240 will automatically word-align it. As a result, **`mem[291]=7734`** has the same exact effect as **`mem[290]=7734`**.

You cannot currently set a range of memory values with one command. So sorry.

10. RISC240 Synthesizable Processor

A RISC240 processor has been described in synthesizable SystemVerilog, which can be programmed onto the Altera FPGA boards. The processor has an FSM which controls the datapath. The datapath contains a variety of components, including an arithmetic logic unit, a register file, and multiplexers. The source can be found in the following files, which are located at `/afs/ece/class/ece240/bin/RISC240`:

File	Contents
alu.sv	the arithmetic logic unit for the RISC240 processor
bram*	Three files are used to instantiate a block RAM (BRAM) for the program memory. The **`memory.mif`** file will be used to initialize the values in the memory.
constants.sv	definitions of readable names for bit patterns like ALU operations, microcode operations, control points, etc.
controlpath.sv	the FSM (i.e., control path) that drives the datapath
datapath.sv	a collection module where all the components in the datapath are instantiated
library.sv	a collection of parameterized components
regfile.sv	the register file, with eight 16-bit registers, and with register 0 hardwired to the value zero

RISC240.sv	the top-level module, which instantiates the datapath, control path, and I/O modules, including an I/O module for simulation

These files will allow synthesis, with a few caveats:

- At the top of **`constants.sv`**, there is a comment (**`\\`define synthesis`**) **that you need to uncomment before synthesizing**. Make sure to re-comment it for any simulation tasks.
- These files have no method for getting input to your program, nor receiving output from it. Lab 6 generally requires creating some memory-mapped I/O ports and connecting them to LEDs and switches.
- Machine code needs to be included in the synthesis run. The **`as240`** assembler outputs a memory image file (named **`memory.mif`** by default, not the **`memory.hex`** file). This file is a memory image of your program. If it is included in the project, Quartus II will find it during synthesis and use it to configure the memory.
- Program memory starts at address $0000 and ends at $03FF. Locations $0400 through $07FF are for data (variables).

Acknowledgments

This manual is based off of the *P18240 Reference Manual*, version 1.2.0.

Especially in the early versions, this document may have typos and other bugs. Please report them on `https://www.piazza.com` or via email to the staff list (`ece240-staff@lists.andrew.cmu.edu`).