

Project RISC

Out: 2 Oct 2009

Helpsession: 5 Oct 2009 7:00PM, Motorola Room

Design Check: 7-9 Oct 2009

Due: 16 Oct 2009, 10:00pm

0 Motivation

You've seen different kinds of computers in the Moon family, learned how they work, and how to change them. Now you're going to build your own RISC, or reduced instruction set computer, from the ground up. We provide you with hardware specifications and an instruction set; you'll come up with a computer.

You have all the tools necessary to do this assignment; it is now a matter of putting them together and making it happen. And in the end, you will have a computer that follows the machine instructions you give it.

1 Requirements

- Sign up for an appointment for interactive design check grading, and show up on time. Sign up sheets are hanging on the wall outside the fishbowl TA room starting today, Friday October 2. You can find more details on what will be required for the design check at the end of this handout.
- Hand in a circuit file and control ROM(s) that realize the hardware specification described.
- Hand in a README file that
 - describes the bugs in your circuit (or asserting their absence)
 - lists and explains your circuit's control wires

2 Details

2.1 Hardware specifications

Single cycle. This helps to simplify the design, means that RISC doesn't need to use a central bus, and means that data and code must reside in different memories (as in Moon-1).

Program ROM. Program code in RISC is located in a program ROM (again, as in Moon-1). Each instruction is eight bits in size. A program counter keeps track of the current instruction being executed, and is incremented after each instruction.

Branching. Flow control in RISC is a little more complicated than in the Moon family of computers. Instead of specifying an absolute address to jump to as in Moon-1, RISC branches *relative to the program counter*. Refer to the JGZ instruction for more detail.

Memory. As in Moon-1, data in RISC is stored separately from the program code. Each word of memory is eight bits in size, and there are 256 words total. References to memory are made *indirectly*, meaning that memory operations get their address from a register. Refer to the section **Use of memory** below, and to the LW and SW instructions for more detail.

Four eight-bit data registers. RISC has four data registers instead of the single accumulator from Moon. Instructions manipulate values in registers, or move information between a register and memory. This adds the complication of specifying which of the four registers to use in an instruction.

The registers are named r0, r1, r2, and r3, and are identified respectively by the binary values 00, 01, 10, and 11, adding a leading zero when necessary.

2.2 Instruction set

Eight instructions. RISC has eight instructions: two are arithmetic operations, two perform movement in registers, two are memory operations, one is a comparison operation, and one is a flow control operation.

Most of the instructions take information from a source, perhaps manipulate it somehow, and put it in a destination. Refer to the instruction list in the next section for more details.

Use of memory. All of the arithmetic and movement operations in RISC are non-memory, which means the source is always an immediate value or a register. The destination is always a register. Therefore, the only way to manipulate data in memory is to transfer it to one of the registers, use it, then put it back in memory.

Consequently, RISC needs only two instructions that deal with memory.

As mentioned previously, RISC references memory indirectly. This means that when reading from or writing to memory, the memory address being touched is not specified explicitly in the command itself. Instead the load and store commands specify a register, which contains the desired memory address.

For the register-to-memory operation, the source register contains the address of the memory location to load into, and the destination register contains the data to load.

For the memory-to-register operation, the source register contains the address of the memory location to read from, and the destination register is where the data from memory will be written.

This is potentially confusing, so make sure you understand the semantics of the memory operations before you continue.

2.3 Instruction layout (RISC machine language)

An instruction consists of an opcode and an operand. The first three bits form the opcode and specify one of the eight instructions. The remaining five bits make up the operand.

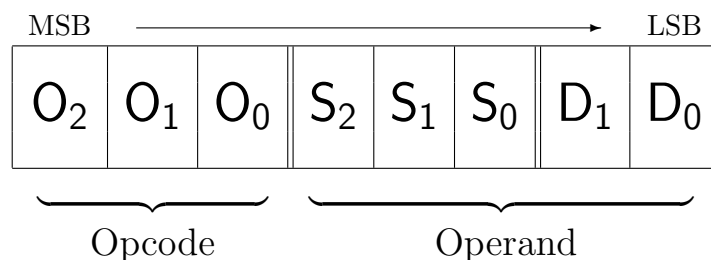
The final two bits always specify a destination register. The first three bits of the operand specify a source register, or, for `MOVI`, an immediate value.

Register specifications are binary numbers from zero to three (as defined earlier), sometimes with a padding zero. `r2`, for example, is identified by `10` (when used as a destination register) or `010` (when used as a source register, since three bits must be filled).

For memory operations, the last two bits of the operand specify a destination register (which holds the data read or to be written), and the first three bits specify the source register (which holds the address where data is read or written).

For MOVI, the last two bits of the operand specify a register, and the first three bits specify a three-bit two's complement immediate value.

For all other operations, the final two bits of the operand specify the destination register, and the first three bits specify the source register.



Note that instructions of the form `XXX rY, rY`, for example `ADD r0, r0`, are legal and should produce the same result as the following two instructions without writing to r1.

```
MOV  r0, r1
ADD  r1, r0
```

2.4 RISC Instructions

LW *src, dst* – Load Word

Opcode: 101

LW moves a word of memory into the destination register. The memory address is specified by the value in the source register.

SW *src, dst* – Store Word

Opcode: 110

SW moves the value of the destination register into memory. The address is specified, as in LW, by the value in the source register.

ADD *src, dst* – Register Add **Opcode: 000**

ADD adds the source register to the destination register and stores the result in the destination register. The source register is left unchanged.

SUB *src, dst* – Register Subtract **Opcode: 010**

SUB subtracts the source register from the destination register and stores the result in the destination register. The source register is left unchanged.

MOV *src, dst* – Register Move **Opcode: 011**

MOV stores the contents of the source register in the destination register. The source register is left unchanged.

MOVI *imm, dst* – Register Move Immediate **Opcode: 100**

MOVI stores a 3 bit two's complement immediate value in the destination register, sign extending if necessary. No other registers are changed.

CMP *src, dst* – Register Compare **Opcode: 001**

CMP is used to compare the values in two registers. If the two numbers are equal, then the value 1 is stored in the destination register. Otherwise, the value 0 is stored.

JGZ *src, dst* – Jump if Greater than Zero **Opcode: 111**

JGZ is RISC's only flow control operation. If the two's complement value of the source register is greater than zero, the program counter is adjusted by the value of the destination register. Otherwise, the program counter is incremented as usual.

2.5 Memory instruction example

Suppose *r1* contains the value 37. `LW r1, r3` loads the contents of memory location 37 into register *r3*.

Again suppose `r1` contains the value 37. `SW r1, r3` loads the value in `r3` into memory location 37.

2.6 Branch instruction example

In the following example, `ADD r2, r3` and `LW r3, r0` are executed repeatedly as long as `r0` is greater than zero. Once it isn't, execution continues with the `MOV r3, r0` instruction.

```
.
.
MOVI  -2, r1
ADD   r2, r3
LW    r3, r0
JGZ   r0, r1
MOV   r3, r0
.
.
```

Note also that the equivalent of a `JEQ` operation can be achieved by combining `CMP` and `JGZ`. For example, the following instructions will branch a number of lines equal to the contents of `r2` if `r0` is equal to `r1`.

```
.
.
CMP   r0, r1
JGZ   r1, r2
.
.
```

2.7 Finishing execution

In testing, sometimes you'd prefer that you didn't keep executing meaningless 00000000 instructions when you reach the end of your program.

One way to achieve an infinite loop that doesn't do anything is:

```
.
.
```

```
MOVI 1, r2
MOVI 0, r3
JGZ r2, r3
```

Note that this does overwrite two registers, so do this only if your testing doesn't require you to see those values.

3 Support

Use Logisim! Type `cs031_install risc` to install the logisim stencil. This consists of an empty control ROM, and links to test programs and to an empty ROM file. It also contains a `template.circ` file, which gives you access to a circuit that tests if all wires are zero ¹.

If you want to use diglog, install the diglog stencil with `cs031_install risc.diglog`.

You should make a copy of the empty ROM to use for your own program ROM files (e.g. `cp zeros.ram mytestprog.ram`). For help turning your assembly code into a working ROM file, read the `romREADME` included with the stencil, which provides detailed instructions and examples.

We're providing two sample programs that you can use to test your computer. These programs do not test everything, and are only meant to ensure that consistent mistakes in your instruction layout don't break your machine. For each sample, you'll see a `.ram` file (to load into your program ROM) and an assembly file that details the instructions and traces register values.

If you need a reminder of how anything works in Diglog/Logisim, then check out the Tutorial and Helpession slides linked from the Docs page on the course website.

For Diglog, there is also a reference file that is installed that explains some components. It includes conventions for which input should be the least significant bit for many components. You are not required to follow these conventions. However, they may help you avoid confusion.

¹Diglog has one built-in, but LogiSim doesn't, so we're providing it for you

4 Extra credit

Given that there are three bits for the opcode and eight instructions in the spec, it's hard to build in any new functionality while maintaining compatibility with the existing machine language.

However, if you have extra time and energy, we'd encourage you to spend time simplifying and condensing your circuit. We will offer up to 3 points of extra credit for handins that have particularly good layout. You will find this task much easier if you put lots of thought into your block diagram for the design check.

5 Grading criteria

- **Design check grading** – 20%
- **Final grading** – 80%

5.1 Late policy

Late projects will be accepted with a 20 percent penalty per day late. Projects handed in more than three days late will not be graded. However, you must hand in a working version of all projects (even if it goes ungraded) in order to pass this course.

Project design checks must be done on the dates specified; late design checks will not be accepted.

6 Handing in

To hand in Logisim RISC, type `cs031_handin risc`. For Diglog, type `cs031_handin risc_diglog`.

You will be prompted for your

- README file
- circuit file (a `.lgf/.circ` file)
- control ROM (`.ram` for Diglog, `.img` for Logisim)

- additional control ROM (another .ram/.img file)
If you only have one control ROM, just press enter when prompted for the additional ROM.
- Test cases (.ram/.img)

If you want to change your handin, just run the script again and the new will supersede the old; we actually won't see anything but the most recent one. Make sure to do this before the due date, though, as changing your handin after the due date will cause it to get marked as late, even if you originally handed your assignment in on time.

7 Final words

Don't gate the clock. As we've mentioned before, gating the clock is **evil** and is therefore against the rules. Do not run the clock signal through any gates before attaching it to elements requiring a clock signal.

You may want to use a negative clock input with some of your gates. To accomplish this, we will allow you to run the clock wire through a single inverter if you are using logisim (if you are using diglog simply use the second clock output).

Also, pay attention to which side of the clock edge (rising or falling) your RAM is triggered on. Either will work, but you will have to be consistent with the rest of your circuits to make sure information from/to the RAM flows as it should.

Your circuit may not contain more than one clock.

Register displays. You must put hex displays (7SEG on Diglog, Hex Digit Display on Logisim) on the PC, the program ROM, and all four registers so that your grader (and you!) can easily see their values. Note that the 7SEG component only takes four bits, so you will need to use two in order to display the value of a byte. When using the 7SEG, in Diglog, you can use either the left or bottom inputs, but remember that in its default orientation the bottom input on the left side and the leftmost input on the bottom represent the most significant bit.

Testing. Don't set your clock speed too high when running your test programs in Diglog. Diglog has a bug (what!?) that leads to incorrect circuit simulation if the clock is too fast. Don't set it faster than 50 cycles/sec. Alternatively, substitute a switch for your clock if your test programs are short.

Design notes. Leave space between your components. Things can get very crowded very quickly.

Although RISC is not a central bus computer, it would be extremely cumbersome to use multiplexers to do all the data path selection. You should use a few smaller buses to simplify things (i.e. replace large arrays of multiplexers with a few tri-states and a bus).

In short, excessive use of multiplexers will be frowned upon.

Logisim has some gates that you are not allowed to use. In particular, from the Arithmetic category, you are **only allowed to use the adder**. You **cannot use the Subtractor or the Comparator** circuits - you'll have to implement them in terms of an adder.

One more thing - the RAM in Logisim lets you specify separate input and output pins. This is **not allowed!** You must have input and output pins go on the **same** wires. These restrictions are in place to make the circuit and control ROM design a little more interesting.

For Diglog, you're free to use any components from the **catalog**. Indeed, proper use of many of these items can help you layout your circuit more effectively. However, you should **not** use any components from the diglog library. If you really feel there is a gate in the library that you need to use, please see a TA on hours or e-mail cs031tas.

If you decide to use Diglog/Logisim on another computer, make sure that the circuit you create can be opened and simulated by the computers in the sunlab. If we cannot open your circuit, then we will not grade it, which means a zero for you. In addition, the restriction on circuit components applies across systems. Again, you should not use any gate that is in the diglog library on the sunlab computers.

Finally, frills are good things. Use lots of labels and boxes in your finished circuit so your circuit components are easy to identify.

8 Design check

Answer the following questions on paper, and be very prepared to explain and defend your answers.

8.1 Block diagram

While there are many different valid ways to design this computer, the handout has hinted that vast quantities of multiplexers is not ideal, and that you would be advised to consider other structures for moving data around (i.e. buses).

Keeping this in mind, draw a block diagram of the organization of your computer. Apply a meaningful label to each major component and bus.

The diagram should evidence significant thought about the project. Vague answers will not receive full credit.

Be prepared to explain each component, what role they play in various instructions, and how they interact with other components. Be prepared to walk through the execution of each instruction.

The block diagram should include all components of your RISC and show how they are connected. There are two differences between your block diagram and your finished RISC:

- the block diagram is done by hand on paper
- your block diagram doesn't have to show all eight wires running from one component to another; one line labeled with a number showing how many wires it represents (that is, draw a slash across the line and write the number of wires the line represents next to the slash) is sufficient.

8.2 Control

How many control wires do you need? How many control ROMs? Each ROM only has eight outputs, so if you have more control wires than that you'll need a second one.

Note that, similarly to Moon-1, each wire should be used for a single function, and there should be almost no gating of control wires with each other.

The idea is that your control ROM should handle the conversion from the opcode to the control wires; having gates as well is redundant and confusing. Please note that decoders and multiplexers are in fact composed of dozens of gates each, and therefore also should not be used on control wires. You will of course need to gate control wires with other non-control wires in logical places. Also, if you've got 9 wires and you really want to avoid using a second control ROM, we'll overlook a well placed gate or two.

For each of your control ROMs, write a short description of what each wire controls, including when it should be high and when it should be low. As an example, let's look at the accWrite line from MOON-1:

The accWrite wire is the write signal for the accumulator. It should be high for those instructions whose results are supposed to be stored in the accumulator, and low otherwise.

8.3 Testing plan

What is your plan for testing your finished RISC? What are the cases you will have to consider?

Simply stated, list what you need to test to convince yourself that your RISC works as it should.

8.4 Test cases

How will you realize your testing plan?

Come up with specific test programs. Then write pseudocode (which you should always do before writing assembly) that tests the things you listed and translate the pseudocode into RISC assembly. Explain what register values you'll need to watch for. Indicate how your test cases cover the various elements of your testing plan. P.S. If you put effort into this now, you won't have to later.