COMS12200 Labs

Week 15: Extending your ISA to 8-bits with memory instructions

Simon Hollis (simon@cs.bris.ac.uk) Document Version 1.0

Build-a-comp module lab format

This worksheet is intended to be attempted in the corresponding lab session, where there will be help available and we can give feedback on your work.

- The work represented by the lab worksheets will accumulate to form a portfolio: whether complete or not, archive everything you do via https://wwwa.fen.bris.ac.uk/COMS12200/ since it will partly form the basis for assessment.
- You are not necessarily expected to finish all work within the lab. itself. However, the emphasis is on you to catch up with anything left unfinished.
- In build-a-comp labs, you will **work in pairs** to complete the worksheets for the first part of the course. In later labs, with more complex assignments, you will work in bigger groups.
- To accommodate the number of students registered on the unit, the 3 hour lab session is split into two 1.5 hour halves.
- **You should only attend one session**, 9am-10:30am OR 10:30am-12pm. **The slot to which you have been allocated is visible on your SAFE progress page for COMS12200.**

Lab overview: Adding load and store instructions to an 8-bit ISA

So far, our machine has had a very small instruction set, but critically only two places to store data during the running of a program. To expand the range of capabilities and the ease of programming, we add

- More instruction
- Data memory instructions

To facilitate this, we will also need to add a data memory component.

As we have seen in lectures, there is a wide choice of ways to access data memory, but we will focus on the simple mechanism of *Direct Addressing*.

Under the Direct Addressing scheme, we supply a memory operation, such as 'load' or 'store', along with the address in data memory to store it to. For example:

"Store r0 in memory location 3" or *"Load memory location 2 into r0"*

Using a common convention of square brackets to denote a memory address, we can define two new instructions for our machine: `STR` and `LDR`. As **an optional extra (do last!)**, also add `JNEG` conditional on a negative result.

However, since we currently use only two bits for our op-code, we do not have room for two more instructions. Therefore, we need to expand the op-code by one bit, making it three bits in length. We'll also take the opportunity to allow four bits of address and expand our jump range. To make things neat, we'll re-define our ISA as an 8-bit ISA, from 4 bits.

Our updated ISA is thus:

Instruction	Op-code	Meaning
INC r	000 r_i 0000	r_i <- r_i + 1
DEC r	001 r_i 0000	r_i <- r_i - 1
JNZ [address]	0100 $a_3a_2a_1a_0$	if (r_0 != 0) then PC <- address ; else PC <- PC + 1
JNEG [address]	0110 $a_3a_2a_1a_0$	if (r_0 < 0) then PC <- address ; else PC <- PC + 1
STR r, [address]	100 r_i $a_3a_2a_1a_0$	MEM[address] <- r_i
LDR r, [address]	101 r_i $a_3a_2a_1a_0$	r_i <- MEM[address]

Adding a data memory

At the moment, our machine has a single memory, used for storing instructions. We call this the Instruction Memory.

Our new Load and Store instructions operate on Data Memory, so we need to add it.

When adding Data Memory to a computing system, we have a choice between implementing it separately, as for a Harvard Architecture, or in a unified manner, as per the Von Neumann Architecture.

For this lab, I would like you to implement your data memory in the Harvard style. Doing it this way:

1. Makes it simpler to create the control path for;
2. Makes it conceptually simpler;
3. Prevents any problems with shared buses and bi-directional transfers.

Task 1: Add a data memory

- Find second memory modules to add to your design. Call the one you added last lab the "Instruction Memory" and the new one the "Data memory".
- The data memory is designed so that it separates out reads and writes on different connectors

Hint: Loads will use the DataOut connector; Stores, the DataIn.

Task 2: Expand your ISA to 8 bits

Earlier in this worksheet, a new 8-bit instruction set is defined. In this task, you need to:

1. Update your prior implementation to take in 8-bit instructions
2. Add the new memory accessing instructions to your control and data paths. Note that the position of some of the bits has changed, but your decoding strategy is exactly the same as before.
3. Add the new `JNEG` instruction *(do this bit last, as an optional extra)*

Task 3: Make use of your new, more powerful machine!

Now that you have an expanded ISA, you can:

- Load and store to 16 data memory locations.
- Jump to any of the first 16 instructions.
- Still increment and decrement

With this power, you can write some more interesting programs!

Experiment with programming the instruction memory with new programs Due to the size of your program counter and your jump immediates, it's probably best to limit these to 16 instructions. You could try:

1. A program that determines if the least significant bit of an input is set. Start with the value you wish to check in MEM[0] and the constant '0' in MEM[1]. Store an input value in the range 0—7 in data memory location MEM[0] (you can store larger, but your program will be >16 instructions!). Write code that checks if the least significant bit is set. It should return '1' in r1 if it is and '0' if not. Your first instructions should bring these constants into registers e.g.

Instruction address	Instruction
0	LDR r0, MEM[0]
1	LDR r1, MEM[1]
2	???
3	???
4	???
5	???

Optional extras:

If you add the JNEG instruction and expand your program counter to 8-bits, you can write larger programs. For example you should be able to:

1. Write a program that calculates the Hamming Weight of an input number (Hamming Weight = the number of bits in the input that are '1').