COMS12200 Labs

Week 13: Building a counter machine – Part 2

Simon Hollis (simon@cs.bris.ac.uk) Document Version 1.2

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

This week, we're going to complete our Counter Machine. As a reminder, we are aiming to build the following Counter Machine:

- A two register Counter Machine
- A 4-bit register length machine
- A machine with the following three primitive instructions:
  - **`INC r (Increment)`**
  - **`DEC r (Decrement)`**
  - **`JNZ a (Jump if r0 != 0)`**
- A source of instructions, which will be switches.

Instruction encoding

Our counter machine has only three instructions, so we can use a very compact instruction encoding. We can make an arbitrary choice of encoding, but I suggest the following makes life easiest:

Instruction	**Op-code**
INC r	$000r_i$
DEC r	$010r_i$
JNZ address	$10a_1a_0$

Where "$00r_i$" means "`00`" followed by the register index, which in a two-register system is either `0` or `1`, so `000` means **`INC r0`** and `001` means **`INC r1`**, and so on.

`JNZ` always tests the value of **`r0`** and has the following effect:

- If **`r0 !=0`**, set the program counter to the value **`a1a0`**, do not alter **`r0`** and **`r1`**.
- If **`r0 = 0`**, increment the program counter as normal; do not alter **`r0`** and **`r1`**.

Task: Build a Counter Machine instruction input and program counter

Last week, you completed the data path of a counter machine. With this, you are able to feed in a particular instruction and execute it.

This week, we will add the ability to process multiple instructions, and be able to keep track of them using the program counter.

As part of the construction of the program counter mechanism, we will build support for the jump (`JNZ`) instruction.

Task procedure:

We need to build the following:

1. A set of input instructions (switches) – use four to start with.
2. A mechanism for selecting one of the available input instructions, based on an address input.
3. A program counter.
4. A clock to automate the execution.

Instruction input

To begin with, take four Input modules to represent four instructions. At any time, only one is needed, so the first step is to create a selectable instruction output, which reduces your four instruction inputs to a single output, based on an address input.

By now, you should be familiar with how to do this in a straightforward manner, so implement this four-to-one mechanism, to create an active instruction output.

Program counter

The program counter is a register that is:

- Initialised to 0 when the Counter Machine is initialised
- Every cycle of execution either
  - Adds 1 to its value *or*
  - Has its value updated to equal the argument of a jump instruction, *if the jump condition is satisfied*.

More formally, if the program counter contains value `PC`, it has two possible transitions, based on an instruction `I`:

```
1. PC = PC + 1 if (I == INC || DEC) || (I == JNZ && r0 == 0)
2. PC = address if (I == JNZ && r0 != 0)
```

Hint: the above represents a logical block that has two possible paths for updating its state, along with a reset signal.

Condition testing

Key to selecting the correct update for the program counter is choice about whether or not the jump's condition is satisfied (i.e. does `r0 == 0`?).

As part of the implementation then, we need to be able to perform this test. Since the registers themselves do not have test functionality, the conditional test must be performed in the ALU.

This means that there is the need for control feedback from the data path to the control path. The comparison result will directly affect the PC's control path.

Clocking

Whilst building and debugging your design, it is advisable to attach a fanned-out clock generator to your register elements, and leave it set to manual mode. This will allow you to advance the machine step-by-step by pressing the manual button.

Once you have completed the design, setting the clock to automatic will allow you to step through the input instructions much faster.

Progress checkpoint

Before going any further, complete the instruction selection and program counter functionalities. If anything is unclear, or you are having difficulty implementing all the required functionalities, call over a demonstrator.

It is important to have the above solidly implemented before going further.

Task 2: Running your first Counter Machine program

Now that you have a working counter machine, it’s time to use it in anger!

In its current incantation, you have only four instructions available, but we can still do something interesting with this!

Question: What does the following program do, and why?

Instruction address	Instruction
0	INC r0
1	JNZ 0
2	DEC r0
3	JNZ 2

Encode the above program and explore its behaviour. If you need assistance, call over a demonstrator.

You now have a working two register Counter Machine, congratulations!