COMS12200 Labs

Week 12: Building a counter machine – Part 1

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 start to build a Counter Machine. The concept was shown in lectures, so should be familiar.

We will 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	**`000`**$\mathbf{r_i}$
DEC r	**`010`**$\mathbf{r_i}$
JNZ address	**`10`**$\mathbf{a_1a_0}$

Where "**`00`**$\mathbf{r_i}$" 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 data path and control

Your job is to build this machine from the Build-a-comp modules. The good news is that the design builds on the work you have done in previous weeks, and you will be able to re-use a good amount of your design so far.

The main additional component you may need over the ones you have used so far is the Fan Out module.

Fan Out module

The Fan Out module is the simplest of the Build-a-comp modules: it simply replicates the input signals and copies them four times. Each four bit output is equal to the input four bits. i.e.

Input	Output A	Output B	Output C	Output D
DCBA	DCBA	DCBA	DCBA	DCBA

Task procedure:

You will need to build the following components and this will take two weeks.

This week we aim to produce.

1. An arithmetic system that can take data from one of the registers and that commits results to the same register.
2. An instruction input unit (a switch)
3. An instruction decoder, which takes the next instruction to be executed and creates the relevant control signals for the machine.

   We will decode, but not implement the function of the jump **(`JNZ`)** ***this week;*** we will leave this to next week. Therefore, this week, we only wish to be able to support the **`INC`** and **`DEC`** instructions.

Data path overview:

- There are two data storage registers.
- Either one can be incremented or decremented, so they need to share a common arithmetic unit.
- The same register that is the data source will also be the data destination.

- You will need to ensure that there are no fast data loops in your design.

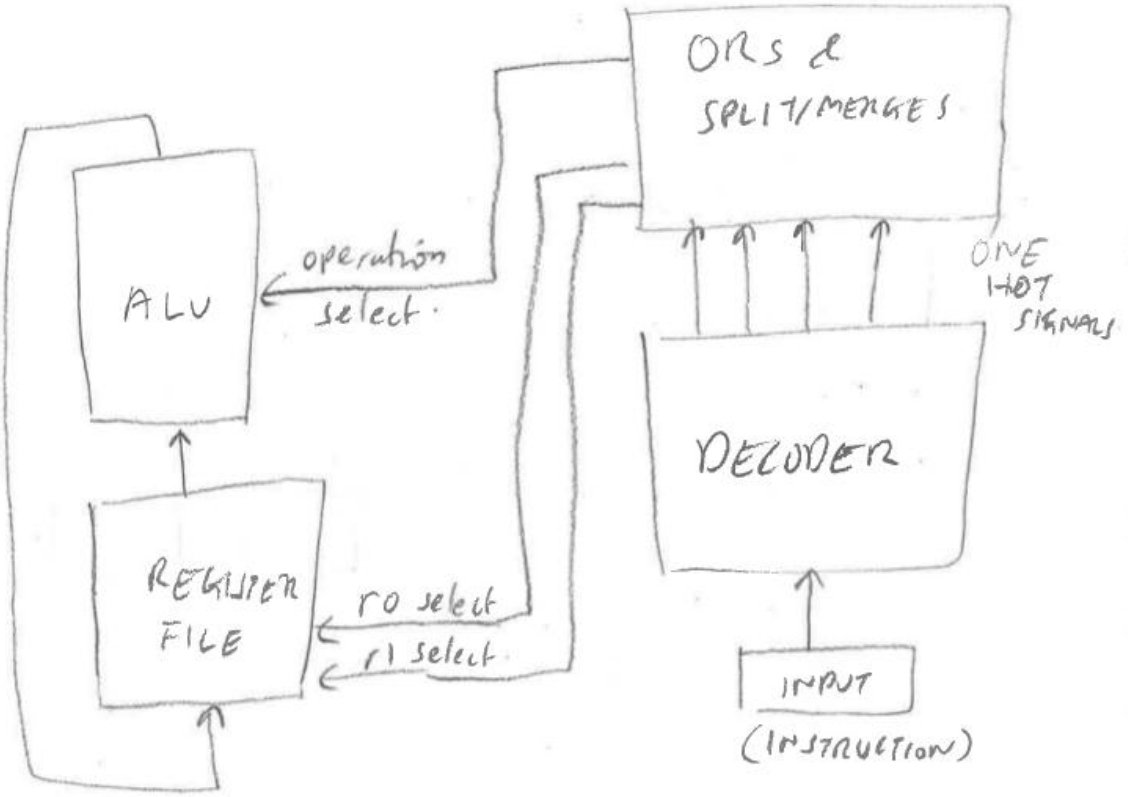
Control path overview:

The control path should be created using the one-hot decoding strategy that you familiarised yourself with in the previous lab. Doing this will provide you with an easy to create and generic mechanism for creating all of the control signals that your data path components need.

Your task is to:

1. Decode the relevant parts of the instruction which determine the operation.
2. Decode the relevant parts of the instruction which determine the register to be used.
3. Use an OR-based structure to generate individual control signals.
4. Use a Split/merge module to merge individual control signals where necessary.

In diagrammatic form, this looks like this (key control path signals are shown; your implementation may need additional control signals).



As before, your best approaches is to create a Boolean expression for each desired control signal, using DNF, then to create the relevant OR structures.

Implementation guidelines

- For this week, use a single Input module to input an instruction.
- Use the manual feature of the clock to step through an instruction, before changing the Input module to input the next one.
- Each bit of the instruction op-codes has a specific meaning. For example, the two MSBs denote whether we will be performing an arithmetic operation or jumping and the two LSBs denote the register in use in an arithmetic operation.
- You will need to use the Split/Merge module to separate out the various control bits from the instruction. You will also need to use it to combine control signals where a module requires both a clock and an enable signal.
    - *Hint: the enable signal(s) will come from the Decode circuitry and the clock signal will come from the Clock module.*

- The Split/merge may or may not give you exactly what you need. If it does not, often adding a shifter on the input or output can align the bits as necessary.
- *We will decode, but not implement the function of the jump* (`JNZ`) *this week;* we will leave this to next week.

Testing your design

When complete, your design should be able to execute an instruction by setting the input switches so that they match a valid op-code and manually clocking the system.

To clock one cycle, set the clock to manual mode and press the "Manual" button four times. This will advance both phases of the clock by one complete cycle and simulate a full clock cycle of the completed processor.

Experiment with the implementation by inputting short programs and observing how the two registers are affected by the INC, DEC and JNZ instructions. Ensure that they operate as expected, so that when you expand your system, there are no hidden surprises!

Building on the design

Next lab, we will extend the system to include a program counter that is updatable via the Jump instruction, so ensure that you have your solution to this week's lab written down.