DECA Lab

Part 4: MU0

Department of Electrical and Electronic Engineering

Imperial College London

v1.3

Spring 2020

Contents

Introduction

Weeks 4 and 5 of DECA lab this Term, described on this sheet, are your first attempt to put together working digital circuit of significant size. You will design a MU0 processor as described in the lectures. You have been well prepared for the actual design work by previous labs and lectures. Still, this is a much bigger design, it will have its own challenges in understanding and testing. The design work is split into two blocks where the first block (the Control path) can be tested independently of the second block (the Datapath). The first block's work is slightly longer, and you will be less practiced, so will probably overflow into the second week.

Quartus has various issues which the lab demonstrators have been helping you with. This term, to allow your design work to go more efficiently, we will detail on Piazza the various issues and recommended workarounds. It is well worth checking this, and asking when you have some problem. Unlike the lab, Piazza will work 24/7!

You have learnt in previous labs:

- to design D flip-flop registers out of `DFF` components.

- to design a state machine using a combinational `next_state` block and a register to store the state.
- to create a ROM from a Quartus component with data defined in a `.mif` file
- to join blocks together on a schematic with busses
- to implement combinational logic on busses with Verilog: also to use Verilog to conveniently combine or split parts of busses

You will use these techniques in this lab, with a Quartus RAM block replacing the ROM you used previously. You will be working with mostly 16 bit busses where combining individual flip-flops to make registers, or 1 bit multiplexers to make 16 bit multiplexers, is not practical. Instead you will use Quartus megafunction blocks that represent whole (16 or 12 bit) registers, or bus multiplexers.

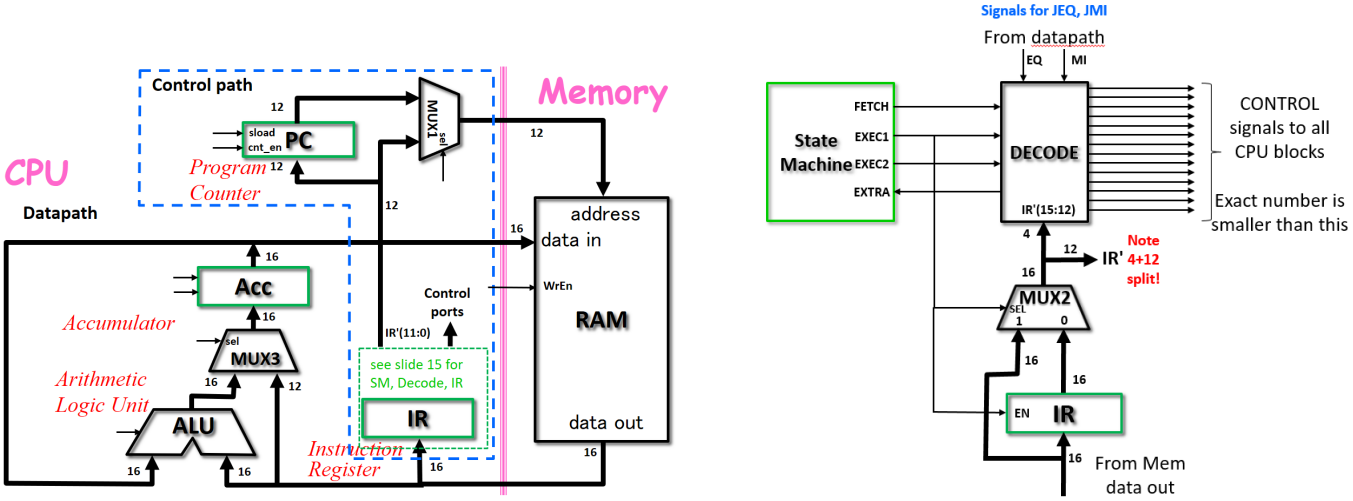
Verilog combinational logic will be needed, for the sometimes complex decode logic. In addition, for convenience, a megafunction ALU block will be used.

So for this lab:

- **Don't use individual flip-flops**
- **Don't use gates as blocks - all combinational logic will be either megafunction blocks or Verilog**

Designing hardware at a higher level will make your life easier.

1. MU0 Control Path Design



slide 5 slide 15

Figure 1: MU0 CPU and memory unit schematics

Before the lab

Your task in the next two labs is to make a Quartus block schematic CPU and memory unit as in Figure 1. The first task is to implement the Control path and RAM. At the end of this lab you will have instructions that can be FETCHed, and JMP instructions that work.
For the control path you will need the blocks as shown in Figure 2. Each block (there are two BUSMUX blocks) corresponds to one or more of the blocks in the control path as shown in Figure 1 which details slides 5 and 15 from the MU0 hardware lecture. Note that you need a total of 3 Quartus pre-written blocks from slide 5, and 2 blocks from slide 15. In addition you will (later) design the State Machine and Decode blocks. The pre-written blocks are ones you already know: DFFs (with enable), counter (with load), MUX, RAM (similar to ROM but can be written).

- Note what control signals are available for each of your 5 blocks (there are 6 signals altogether) and what function they serve in the overall schematic. For example, the `MUX2` input `sel` switches the PC output (in state FETCH) or the IR' bits (when accessing location 'N') onto the memory address lines.
- The boolean logic for the IR and its MUX (MUX2) are given to you. By tracing memory data in states `EXEC1` and `EXEC2` work out the reason for the `MUX` and why `IR'` and not `IR` is used to control the rest of the design.
- Where you don't know what a control signal does, make a note to ask demonstrators in the lab.
- Read Section 1 of this handout briefly so that you understand what you will be working towards in the session.

Clock	Symbol	IP	Data ports	Control ports	Function	Name
No	gates/busmux		dataa, datab, result	sel	multiplexer	MUX1, MUX2
Yes	storage/lpm_ff		data, q	sload	DFF register	IR
Yes	arithmetic/ lpm_counter		data, q	cnt_en, sload	loadable counter	PC
Yes		ram 1-port	address, data, q	wren	read/write RAM	RAM

Figure 2: Blocks for Control path: see Figure 3 for control port operation

Device	Input	Operation	1	0
busmux	sel	Select	datab	dataa
lpm_ff	sload	dff load	enabled	disabled
arithmetic/lpm_counter	sload	counter load	enable	disable
	cnt_en	enables +1 count	enable	disable
RAM 1-Port	wren	operation	write	read

Figure 3: Block control signals with operation

1.1. Implementing the Control Path schematic blocks

Look in Quartus using the add symbol tool, under `megafunctions`, to find the blocks shown under the *symbol* column in Figure 2.

Note these megafunction blocks are mostly also available under IP catalog - but the advantage of adding them as symbols is that it is faster.

Using the IP Catalog window you will need to add the one block shown under Ip column. The procedure for adding this to your schematic is identical to the ROM you used in lab 2. In the ADD dialogue change settings as in Figure 4.

Megawizard	Change from default
page 1	RAM width in bits, RAM size in words
page 2	untick q output port registered
page 5	Select "no leave it blank" for the memory data
page 6	Tick ram.bsf

Figure 4: RAM configuration

□ **Task 1.** *Place the blocks from Figure 2, as required to implement the control path schematic on a new schematic sheet (MU0CPU). If you get this wrong you can always add/delete later.*

The megafunction blocks placed using the symbol tool can have properties configured by right-clicking on the block and selecting *properties*. In the properties dialog you must set:

- Instance: choose a suitable name for each block, e.g. PC for program counter etc.
- Ports: set to `unused` the ports you do not need. All data ports shown in Figure 2 are used. Figure 3 gives the control ports you need: in addition clocked blocks need clock. All other control ports can be set as *unused*. Note that ports starting with '`a`' are for asynchronous operations and will not be used.
    1. The defaults values for unused ports are always the correct (constant) value for the block to work as expected.
    2. All the ports starting 'a' relate to asynchronous operations you will never use, make them unused. Ports starting s are for clock synchronous operation, and will be used if needed.
    3. All blocks have an `enable` port that you do not need to use.
    4. In addition most have synchronous set and clear ports (which set the storage to all zeros or all ones). You do not need to use these.
    5. Note that by default all block flip-flops are initialised on power up to 0. You therefore do not need to implement a specific reset signal.
- Parameters: set width to the required bit-width of your block. Do not change anything else.

□ **Task 2.** *Configure the properties of each of the blocks you are using, and connect up your megafunction block busses as per the schematic. Where an input bus comes from the Datapath, not yet implemented, connect it to GND so that the circuit will simulate. See Appendix A for help in connecting busses and control signals.*

1.2. Implementing the State Machine

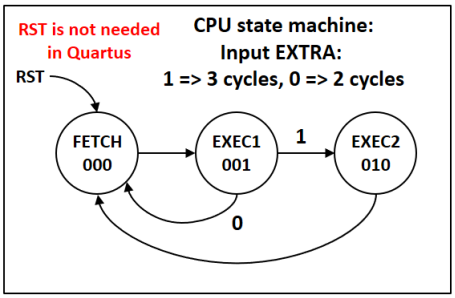


Figure 5: MU0 State Machine: `RST` is not needed because Quartus initialises all registers to 0

The required boolean logic for `NS(2:0)` is very simple and can be determined from Figure 5 - the state machine shown in lecture 4 - slide 12. It is not not necessary to have additional logic to generate `EXEC1` and `EXEC2` outputs because they are available as `S(0)` and `S(1)`, however you can output these signals from the combinational block with the correct name, using a Verilog assignment. That also gives you some flexibility later on if you wish to change the state machine.

□ **Task 3.** *Add your state machine to the schematic as a 3 bit register (for state) and a block of combinational logic implementing the `NS(2:0)` bus and the `FETCH, EXEC1 and EXEC2` outputs.*

There are a total of 7 signals needed as output from the decode block, of which two are given to you in Figure 1. Working out these signals is the most difficult part of the lab. Each signal must be correct for each instruction, and in each state (a total of 3 times 10 cases). However, most signals simplify. For example the `EXTRA` output to the state machine must be 0 or 1 according to whether the instruction (in `IR'`) needs 2 or 3 cycles. No timing information is required, and this can be decoded from just the 4 `IR'` bits.

No register is written during `FETCH`, and `MUX1` must send `PC` to the RAM address. In `FETCH` all other MUX inputs are don't care. Only a few instructions need 3 cycles and therefore use `EXEC2`. It is also helpful to write the signals in an intermediate form: `EXEC1&(STA + LDA) + EXEC2&(LDA+ADD+SUB)` rather than in terms of of individual bits. See Appendix C for how to do this in your Verilog code.

□ **Task 4.** *Add another combinational block called `DECODE`, as in lecture 4 - slide 15. Implement boolean logic for the all necessary control signals in Verilog in this block.*

1.3. Adding a test program and simulating

In order to simulate this hardware you need to define RAM initial contents, setting up a program that will execute from address 0. You can do this in whatever way is easiest. For example, you could use the GUI MIF creation method from lab 2. Or, you could write a program in Visual2-deca, run *test-¿ add MIF*, and follow Appendix B to upload your MIF file to Quartus (NB note Piazza).

□ **Task 5.** *Add a simple test program containing the code in Figure 6 and check that the control path works with normal and JMP instructions.*

Challenge

The challenge here, if you have time, will be to add your own instructions. That will be at the end of the second session, so if finish early you might want to start that.

```
LDI 5
LDI 3
JMP 4
LDI 0
LDI 1
STP
```

Figure 6: Code for a first simple test

2. Datapath Design

The work in this section is very similar in structure to that in last week's Control Path design lab. It should be shorter, so if you have not yet finished last week do not despair!

Before the lab

Read lecture 4 - slide 5 and MU0 Timing.

- For the Datapath section of the CPU in Figure 1 additional blocks that you will need are a shift register: `megafunctions/storage/lpm_shiftreg`, and an add/subtract block: `megafunctions/arithmetic/lpm_add_sub`, and a multiplexer `megafunctions/gates/busmux`.
- Note what control signals and parameters each block needs to operate as required, see Figure 7. Note that only `Acc` requires a clock: that ALU is combinational logic. Work out, for each instruction, what boolean equation drives each control signal when things happen (usually `EXEC1`)
- The LPM_SHIFTREG block is configured to shift right and will not on its own implement `LSL`. Leaving this instruction unimplemented is acceptable because it can easily be replaced by `A := A+A` which requires only two instructions. In order (optionally) to implement it you will need to add another multiplexer on the input of `LPM_SHIFTREG`, selecting shifted output or `MUX3`, to turn this register into a shift right or shift left or load register.

2.1. Implementing the Datapath schematic blocks

The schematic does not include every required signal, though it does indicate most. For example bits 15:12 of the memory data out input to `MUX3` are not shown. These need to be memory data out bits 15:12 usually, but will normally 0 in an `LDI N` instruction when it is required to load `N` (12 bits only) into `A`. Also note that some additional combinational logic, not shown, is needed (e.g. to generate the `EQ` signal, and the `MUX3(15:12)` inputs as above).

□ **Task 6.** *Add the blocks needed for the Datapath to your schematic*

□ **Task 7.** *Configure the properties of each of the blocks you are using, and connect up your mega-function block busses as per the schematic.*

□ **Task 8.** *Implement boolean logic for all of the necessary Datapath control signals in Verilog, either in your existing DECODE block, or in some other new block. Include logic for the `MUX3` data input signals, bits (15:12), that are missing from the schematic*

Device	Input	Operation	1	0	Block
storage/lpm_shiftreg	en	enables shift or load	enable	disable	`Acc`
	sload	enables load	load	shift	
	shiftin	shift input	1	0	
	`LPM_DIRECTION`	`"RIGHT"`			
	`LPM_WIDTH`	16			
busmux	sel	Select	datab	dataa	`MUX3`
arithmetic/lpm_add_sub	add_sub	operation	ADD	SUB	`ALU`
	`LPM_WIDTH`	16			

Figure 7: Datapath block control ports and parameters

□ **Task 9.** *Check that you have now connected everything. All signals previously connected to 0, because the Datapath was not implemented, should now be connected. No input should be left unconnected (if it is, the circuit will not compile).*

2.2. Adding a test program and simulating

1. Test instructions one by one.
2. In order to test the Datapath you will want to set up some memory locations so you can see the effect of memory read, write, add, subtract etc. You can do this with DCW and ORG as shown in Figure 8.
3. The most complex instructions are JEQ and JMI, which use data from the accumulator to determine whether the branch happens. Test these last.
4. Remember to keep detailed records of how you have tested your hardware, for the oral.

```
LDA 0x100
ADD 0x101
STP
ORG 0x100
DCW 0x123 // test data for LDA
DCW 0x450 // test data for ADD
```

Figure 8: Code to test LDA and ADD using ORG and DCW

□ **Task 10.** *Add a simple test program and check that the Datapath works.*

Challenge

If you have time, enhance your design in one of the following ways. Make sure you keep your old design for safety and comparison purposes.

- Work out one or more new MU0 instructions - ones that you think will be easy to implement and useful.
- Change your state machine and decode logic to speed up your CPU by pipelining instruction FETCH and EXEC1 phases.

A. Connecting Busses and Wires on a Quartus Block Schematic Sheet

This section summarises and adds to what you were taught in Lab 1 about nodes (thin line 1 bit connections) and busses (thick line multi-bit connections) on Quartus block schematics. There are three ways to connect busses:

- **Direct connection with bus tool**. This will work as long as the two busses are the same width, and is very convenient. Having connected busses you can also name the entire bus (as in 2. below) to allow some of its signals to be connected by name.
- **Connection by name**. You can give any bus a name: e.g. `ALUOUT[15..0]`, by selecting the bus and using right-click properties. Two busses with the same name will always connect, even if on the schematic they are disconnected. You can use this method, connecting a small length of bus to a pin and leaving it unconnected to anything. Name it to connect it to another same named bus on another pin. If you do this you can connect a smaller bus to part of a larger bus by changing the name e.g. `ALUOUT[15..12]` would connect any 4 bit bus to the MS 4 bits of ALUOUT. In fact you can also connect a bus or 1 bit node to a larger bus that contains its signals. This looks neater, and is allowed because it is the names alone that determine connectivity.
- **Concatenation**. Any bus name can be made using concatenation of bus names or signals with ',‘. For example `ALUOUT[15..12],0,0,0,0,SYS[7..0]`. The 16 bits in this bus are named from left to right as given, and therefore connect to other busses with the same names. The 0 named bits connect to the 0 node that you have connected to GND - see below. Arbitrary constant values can be set using *concatenation* of 0 and 1 names: `0,1,1,0` would connect a 4 bit bus to constant `0110`.

Single bit signals (wires) work in the same way. They can be connected by direct node (thin line) connector, or with two small lengths of node connection each labelled with the same unique signal name e.g. `XSEL`. For each connector (bus and node) you can choose diagonal or angle versions as is convenient, and join up any number of segments. Probably not useful, but note that you can turn a bus connector into a node and vice versa by right-clicking.

A useful tip is to create logic 0 and logic 1 nodes named `0,1` respectively and connected to GND and VCC symbols, from the symbol tool: `primitives/other`, as shown in Figure 9. These names work fine, and allow you to connect any bus or node input lines to 0 or 1 by naming them 0 or 1, concatenating that name with others as required.

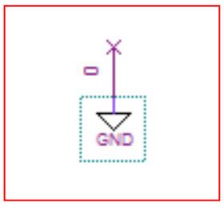


Figure 9: a GND node connection named `0`

```
module adder8_2(op exec1,exec2,fetch,x);
  input [3:0] op;
  input exec1,exec2,fetch;

  output x;

  wire lda, sta, add;

  assign lda = !op[3] & !op[2] & !op[1] & !op[0];
  assign sta = !op[3] & !op[2] & !op[1] & op[0];
  assign add = !op[3] & !op[2] & op[1] & !op[0];

  assign x = sta & exec1 | lda & exec2 | add & exec2; // logic using wires
endmodule
```

Figure 10: Verilog combinational logic
with internal wires `lda, sta, add`

B. Uploading MIF files to Quartus

1. (If using Visual2_deca) Cut and paste the created MIF file from Visual2_deca to a file `something.mif` on your PC, leaving out the `%----MIF----%` lines.
2. Upload this file to your `nfshome` filespace on the server you are currently using. `MobaXterm` has drag and drop file upload via its LH explorer panel when you are connected to a server.
3. Go to your RAM block properties dialog, set the initial data to come from the MIF file you have created.

C. Using Wires in Verilog Logic

In Verilog blocks the `wire` declaration is used to define intermediate signals in your design neither inputs nor outputs. This works like local variables in a program and can be used to simplify the logic, for example as in Figure 10. FPGAs will always optimise logic so there is no cost in this style of implementation.