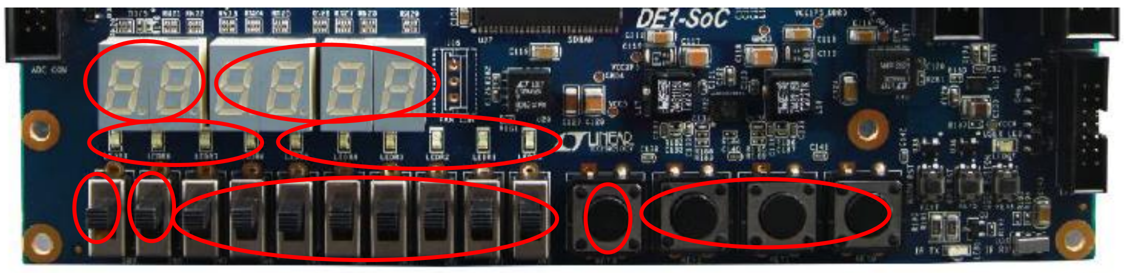
Digital Systems Design 2017 Project 1 Part 1 – Computer

Professor Jonathan H. Manton

Interface

A computer needs to have inputs and outputs to be useful.



Numbered from left to right, top to bottom:

1. Current contents of Instruction Pointer (IP) (hexadecimal)
2. Can be blank, or can display a decimal number between -128 and 127 inclusively
3. Available for hardware debugging
4. General Purpose Output: each LED can be turned on or off
5. Reset switch (HIGH to reset CPU)
6. Turbo switch (HIGH for turbo mode)
7. Data In: will be read by the CPU when the Sample Button is released
8. Sample Button (push then release for Data In to be read into CPU)
9. Buttons: CPU will be notified each time a push button is released

In the above list, items in orange are built into the computer hardware, items in purple are outputs under software control, and those in blue are inputs that can be read by the software. Note though that the software cannot read in the current values of the switches ("Data In") when it pleases. Rather, the CPU hardware arranges for the values of the switches to be latched into the CPU whenever the Sample button is released. This latched version is available for the software to read at any time.

CPU Pins

The Central Processing Unit (CPU) will have the following pins, allowing it to communicate with external circuitry.

Input Pins

Din (Data In)	8 bits	Allows the CPU to read in an 8-bit number. Must be kept stable around the time when the Sample pin goes from HIGH to LOW (falling edge).
Sample	1 bit	On the falling edge, data on the Din pins will be clocked into the CPU.
Btns	3 bits	Three buttons. On a falling edge, a “button release” will be recorded.
Clock	1 bit	External clock (square wave, system-wide)
Reset	1 bit	When HIGH, reset CPU. Minimum pulse duration should be 25 ms.
Turbo	1 bit	When HIGH, CPU will run at full speed

Output Pins

Dout (Data Output)	8 bits	Used to output a number
Dval (Data Valid)	1 bit	HIGH if Dout is valid
GPO (General Purpose Output)	6 bits	Additional output lines for CPU
Debug	4 bits	Used to pass information out of CPU module
IP (Instruction Pointer)	8 bits	Current value of IP register

CPU Architecture

Our CPU is not going to be efficient! Only once we have built our own primitive CPU, can we truly appreciate more refined CPUs.

Note: If you are not familiar with CPUs then you might not be able to guess from this overview exactly how a CPU works. More detail will be given later, if not explicitly then at least implicitly, when the implementation is described in stages.

Registers

The CPU will have access to 32 8-bit wide registers, numbered 0 to 31. Most of them are general purpose registers. The last four registers though have special purposes.

Register 28 (**DINP**) holds the latched contents of the Din CPU pins.

The contents of Register 29 (**GOUT**) will appear on the Dval and GPO output pins of the CPU.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Dval	-	GPO[5]	GPO[4]	GPO[3]	GPO[2]	GPO[1]	GPO[0]
DVAL							

The contents of Register 30 (**DOUT**) will appear on the Dout output pins of the CPU.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Dout[7]	Dout[6]	Dout[5]	Dout[4]	Dout[3]	Dout[2]	Dout[1]	Dout[0]

Register 31 (**FLAG**) is known as the Flag Register. Certain events will cause it to be modified.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	Shift	Overflow	Sample	Btn 2	Btn 1	Btn 0
		SHFT	**OFLW**	**SMPL**			

If a pushbutton is pressed then released, the corresponding bit of the Flag Register will be set to a 1. It will remain a 1 until cleared by the programmer. The Overflow bit will be set or cleared after each arithmetic operation that has the possibility of overflowing, such as addition or multiplication. The Shift bit will be modified after a shift operation has taken place; it is the value of the bit 'shifted out'.

Program Memory

The CPU will read instructions from the Program Memory, which is a block of memory containing 256 memory cells, each cell being 35 bits wide. This memory is read only and asynchronous[1].

Instruction Set

Each instruction is 35 bits wide: such a large width wastes memory but makes the job of both the programmer and designer easier. The programmer can write in machine code more easily due to the simple structure, and similarly, when we come to designing the CPU, it will be easier to decode each instruction. A 35 bit instruction is decomposed as follows.

Bits 34-31	Bits 30-28	Bits 27-18	Bits 17-8	Bits 7-0
Command Group	Command	Argument 1	Argument 2	Address

Command Group

The first 4 bits of an instruction encode the Command Group.

4'b0000	NOP	No Operation
4'b1000	JMP	Jump
4'b0100	ATC	Atomic Test and Clear
4'b0010	MOV	Move
4'b0001	ACC	Accumulate

Command

The next 3 bits of an instruction specify the Command. Their meaning depends on the Command Group. The following types of **JMP** are recognised.

3'b000	UNC	Unconditional Jump
3'b010	EQ	Jump on Equality
3'b100	ULT	Jump on Unsigned Less Than
3'b101	SLT	Jump on Signed Less Than
3'b110	ULE	Jump on Unsigned Less Than or Equals
3'b111	SLE	Jump on Signed Less Than or Equals

[1] A more realistic computer would use synchronous memory, but then we would have to use every second clock cycle to fetch the next instruction in from memory, unnecessarily complicating the design. That said, it is common to use asynchronous memory for controlling FSMs on FPGAs; see Chapter 18 of the textbook.

The **ATC** Command Group is used to test a particular bit of the Flag Register. The 3-bit Command specifies which bit (from 0 to 7).

The following types of **MOV** are recognised.

3'b000	PUR	Pure Move
3'b001	SHL	Shift Left
3'b010	SHR	Shift Right

The following types of **ACC** are recognised.

3'b000	UAD	Unsigned Addition
3'b001	SAD	Signed Addition
3'b010	UMT	Unsigned Multiplication
3'b011	SMT	Signed Multiplication
3'b100	AND	Bitwise And
3'b101	OR	Bitwise Or
3'b110	XOR	Bitwise XOR

Arguments 1 and 2

The next 10 bits of an instruction comprise Argument 1, and the 10 bits after those comprise Argument 2. Arguments describe either an 8-bit number or a location in one of three ways. Decompose the argument as {a,b} where a is a 2-bit number and b an 8-bit number. If a=00 then the number is simply b. For example, 10'b00_0000_1000 represents the number 8. If a location is required, it is an error to use a=00. If a=01 then the location is Register b, and if a number is required, then the number is whatever is in Register b. If a=10 then the location is Register c, where c is the current value of Register b; this affords an extra level of indirection that can be useful to have. Similarly, if a number is expected then the number is whatever is in Register c.

Whether a location or a number is expected will be clear from the Command Group.

Argument (2 bits, 8 bits)	**Location**	**Number**
{ 2'b00, b }	-	b
{ 2'b01, b }	Register b (0 to 31)	Contents of Register b
{ 2'b10, b }	Register c, where c = contents of Register b	Contents of Register c, where c = contents of Register b

Address

Finally, the remaining 8 bits of an Instruction store an Address that is used for the JMP and ATC Command Groups.

Remark

There are many instructions CPUs normally have that we have omitted for simplicity. Nevertheless, this CPU already has more than enough instructions with which to implement an RPN calculator. You can always add more if you wish though!

Stage 1

Download the Skeleton Project directory, rename it, and use it to develop your project. The top-level module is in MyComputer.v.

Go to Assignments > Settings > Files and add MyComputer.v, AuxMod.v, CPU.v, ROM.v and CPU.vh to the project. (The MyComputer.sdc should already be listed there.)

MyComputer.v	Top-level design
AuxMod.v	Auxilliary Modules
CPU.v	CPU module
ROM.v	Program Memory (including program to run)
CPU.vh	`define definitions

Create an empty module for your CPU in the file CPU.v. You must fill in the correct inputs and outputs. All inputs and outputs should be unsigned, and use the standard convention for ordering multiple bits (e.g., input [7:0] Din).

Instantiate the CPU module in the top-level module MyComputer.

Feed the 50 MHz clock into the CPU.

Connect the GPO pins to the six right-most LEDs.

Connect the Debug pins to the remaining LEDs.

Connect the right-most eight switches to Din.

Connect the Turbo switch to the Turbo pin.

Connect **an inverted version** of the left-most push button to Sample. (The CPU expects "1" to mean a button has been pressed, but on the DE1-SoC board, pressing a push button produces a "0".)

Connect **inverted versions** of the remaining three push buttons to Btns.

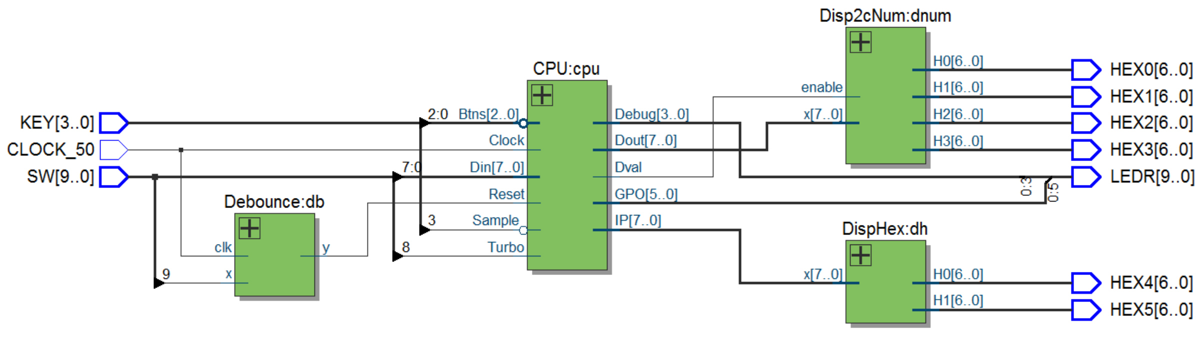
Create a module Debounce (in AuxMod.v) that takes as input a clock (50 MHz) and a wire, and returns a debounced version of the signal on that wire. Stage 2 will create the debounce circuitry; for now, just leave empty the body of the module. Connect the reset switch to the reset pin of the CPU via an instantiation of this Debounce module.

Create a module Disp2cNum (in AuxMod.v). It takes an 8-bit 2's complement number as input and displays the number on four 7-segment displays. A subsequent stage will fill in this module; leave it blank for now. Connect the Dout of the CPU to an instantiation of Disp2cNum, and connect the output of this Disp2cNum instance to the right-most four 7-segment displays. Add an extra input to the module, called "enable". Connect the Dval CPU pin to "enable", so that the CPU can switch off the display if there is nothing to display.

Create a module DispHex (in AuxMod.v) that displays an 8-bit number in hexadecimal on two 7-segment displays. Use this to display the output of the CPU's IP pins on the left-most 7-segment displays. Leave the module blank for now.

Note

The intention is not for you to reverse-engineer any RTL diagrams that are provided. Some of the writing is very small, possibly illegible. They are there simply to give you something to compare with, in broad terms, to see if you are on the right track or not. Ultimately, you will need to test that your circuit works (e.g., by writing test benches). Also, you are encouraged to choose more meaningful names than I do.



Stage 2

External inputs must be synchronised, to avoid metastability. Furthermore, we are told that the reset switch needs debouncing, and of a minimum 25 ms duration. (The other switches do not need debouncing because we will tell anyone that wants to use our computer not to change the switches around the time when the Sample button is released for else the results may be unpredictable. The switches still need synchronising though, just in case our "friends" try to break our computer by madly pressing buttons and moving switches at the same time…)

Synchroniser

Create a module Synchroniser (in AuxMod.v) that takes as input the 50 MHz clock and an input wire, and by using a double flip-flop synchroniser, produces a synchronised version of the input signal. (If you are truly paranoid you can use three flip-flops, but Quartus assures me that we can expect less than one error in 1,000,000,000 years by using just two flip-flops in our design.)

Debounce

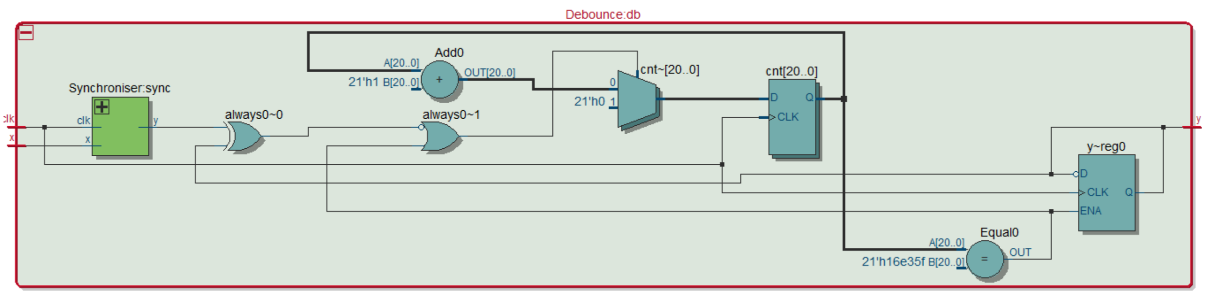
Fill in the Debounce module, so that it operates as follows. First, send the input through a Synchroniser. Then implement a FSM that functions in an equivalent way to the following description. If the synchronised input remains a 1 for 30 ms then output a 1. If the synchronised input remains a 0 for 30 ms then output a 0. Otherwise, do not change the output.

Your design must ensure that the shortest period the output can be HIGH for is at least 25 ms (because this was a stated requirement[2] on the CPU Reset pin specification given to you). In particular, you must ensure there are no glitches on the output. Therefore, you should not use combinational logic or a latch to produce y. Instead, **use a flip-flop**.

Make sure that if the synchronised input changes right at 30 ms, that you do not accidentally change the output to the wrong value (i.e., do not naively make the output equal to the synchronised input once 30 ms expires, unless you are sure you know it will work).

Hint

Here is my implementation of the Debounce module. Remember, you should not try to reverse-engineer the RTL, but rather, write the Verilog code yourself and then compare in broad terms whether the produced RTL is similar to the following. Maybe you have a better design?



[2] This requirement is mainly to give you practice; 25 ms is considerably longer than necessary.

Stage 3

To get something up and running, we will implement a CPU that just does the following.

When the Reset pin is HIGH, all the output pins will be set to 0.

When the Reset pin is LOW, the CPU should run, which in this case, just means setting all the GPO pins to 1.

Simulate your design to see if it works as expected. Make sure to check that the Debounce circuitry functions correctly.

Stage 4

Fill in the modules Disp2cNum and DispHex.

The following module will display a blank, a negative sign, or a hexadecimal digit. (It should already be in the AuxMod.v file that you downloaded.)

```
module SSeg(input [3:0] bin, input neg, input enable, output reg [6:0] segs);
    always @(*)
        if (enable) begin
            if (neg) segs = 7'b011_1111;
            else begin
                case (bin)
                    0: segs = 7'b100_0000;
                    1: segs = 7'b111_1001;
                    2: segs = 7'b010_0100;
                    3: segs = 7'b011_0000;
                    4: segs = 7'b001_1001;
                    5: segs = 7'b001_0010;
                    6: segs = 7'b000_0010;
                    7: segs = 7'b111_1000;
                    8: segs = 7'b000_0000;
                    9: segs = 7'b001_1000;
                    10: segs = 7'b000_1000;
                    11: segs = 7'b000_0011;
                    12: segs = 7'b100_0110;
                    13: segs = 7'b010_0001;
                    14: segs = 7'b000_0110;
                    15: segs = 7'b000_1110;
                endcase
            end
        end
        else segs = 7'b111_1111;
endmodule
```

DispHex is straightforward to write: it just requires two instantiations of SSeg.

For Disp2cNum, note that leading zeros should be suppressed, so that -1 is displayed with two leading blanks followed by -1, rather than -001. If you were writing software, you would use a “for” loop. Because this is hardware, you should instead create a module that can be daisy-chained together, to achieve the same algorithm.

When you believe you have finished, you can test it out by modifying your CPU to include a 23-bit counter. Every time the CPU is running, on the rising edge of the clock, the counter should increment by one. When the counter equals zero, the following registers should be incremented by one: Dout, GPO, IP. Don’t forget to enable Dval.

You should observe the following. When the reset switch is on, only 00 is showing (on the left-most two 7-segment displays). When the reset switch is off, the LEDs should count in binary from 0 to 63, the left-most two 7-segment displays should count from 00 to FF in hexadecimal, and the remaining 7-segment displays should count from 0 to 127 then from -128 back up to 0.

Note: You can use simulations to verify your design. You may also decide to create a new Project Directory for each stage. Then when you get to the Lab to use a DE1-SoC board, you can test out each stage quickly.

Hint

Here is a big hint.

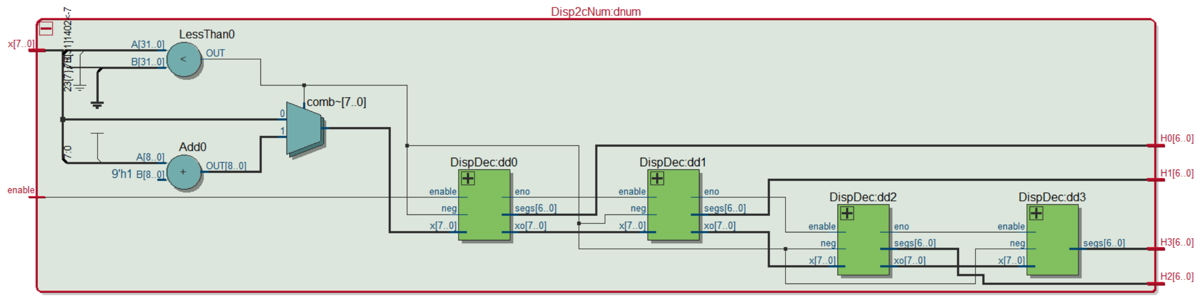
```
module Disp2cNum(input signed [7:0] x, input enable, output [6:0] H3, H2, H1, H0);
        wire neg = (x < 0);
        wire [7:0] ux = neg ? -x : x;

        wire [7:0] xo0, xo1, xo2, xo3;
        wire eno0, eno1, eno2, eno3;

        // You fill in the rest: create four instances of DispDec.
endmodule

module DispDec(input [7:0] x, input neg, enable, output reg [7:0] xo, output reg eno, output
[6:0] segs);
        wire [3:0] digit;
        wire n;
        SSeg converter(digit, n, enable, segs);

        // You fill in the rest. Only a few lines of code are required.
endmodule
```



It may be handy to recall that you can find the least-significant decimal digit using the modulo 10 operation, written "% 10" in Verilog. And you can divide by 10 using "/ 10". So if x is your unsigned number, you can first display "x % 10" and then pass "x / 10" on to the next DispDec.

Stage 5

We will start by implementing just a single instruction. We will not even fully decode it properly. We want to be able to move a number into Register 30 so that it will be displayed on the 7-segment display. (Recall that Register 30 is one of the special registers; whatever is stored in Register 30 will be output on the Dout pins of the CPU.)

The actual instruction should be written as 35'b0010_000_00_xxxxxxxx_01_00011110_00000000 where the xxxxxxxx represents the 8-bit number we want to store in Register 30. Referring to the CPU Architecture – Instruction Set, the first 4 bits (0010) specify the MOV command. The next 3 bits (000) show it is a 'Pure Move' command. The first argument is the source: 00_xxxxxxxx. This represents a number because the leading two bits are 00. The destination of the move is the second argument: 01_00011110. The leading 2 bits (01) indicate the destination location is a register, and the following 8 bits (00011110) represent the number 30, hence the destination is Register 30.

Writing instructions in binary will give us a good understanding of how CPUs work. However, it quickly becomes tedious. A standard trick is to use constants, as now described. It is important to keep in mind that the substitution occurs at compile time: it has nothing to do with how the CPU actually works.

```
`define MOV 4'b0010
`define PUR 3'b000
`define NUM 2'b00
`define REG 2'b01
`define N8 8'd0
```

We can then write `{`MOV, `PUR, `NUM, 8'h88, `REG, 8'd30, `N8}` instead of `35'b0010_000_00_10001000_01_00011110_00000000`. They are identical as far as the Verilog compiler is concerned, but the former is easier to read: we can see immediately that we wish to move the hexadecimal number 88 into Register 30.

We also want to be able to slow the clock down so that we can "see" the computer work on the DE1-SoC board. (In simulations, we want it to run at full speed, hence the reason for implementing a Turbo switch.) Naively, we may think to pass either a slow clock or a fast clock to the CPU module. This is not a good idea though. Instead, the preferred method is to use the Clock Enable pin on flip-flops to slow things down. We will not explicitly tell Quartus to use Clock Enable, but the Verilog code we write will result in the use of Clock Enable. (In fact, the only benefit of Clock Enable is it saves some power; the alternative approach the synthesis tool might take is to feed the output of a flip-flop back to its input via a multiplexer. We could force the issue with a direct_enable attribute, but there is no need. The functionality will be the same.)

Step 1 of Stage 5

Copy your Quartus Project Directory if you like, to back it up. Then delete the body of the CPU module since we will be starting again (but using code similar to what you developed already).

To slow the computer down when required, implement a counter that resets back to zero when it reaches a particular value. Only start a new instruction cycle when the counter is zero. By changing the value at which the counter resets to zero, the speed of the CPU can be controlled. For the moment, choose a value so that a new instruction cycle starts every 250 ms.

Each cycle, simply increment IP by 1, unless the Reset pin is asserted, in which case, set IP to 0.

If you have access to a DE1-SoC board, test your design: the left-most two 7-segment displays should count at a rate of 4 numbers per second, or be reset to 0 if the Reset switch is HIGH. If testing via simulation, change the value at which the counter resets, to a small number like eight.

Note: The simpler design is having a synchronous Reset. The requirement that the reset pin is HIGH for at least 25 ms is not needed. It was included for practice because some devices really do have a minimum pulse-width requirement. (It is still very important that the Reset pin is synchronised to the 50 MHz clock via the Synchroniser.)

Hint

For the counter, use something like the following. (You can add the bit widths to the constants, once you figure out how many bits are required.)

```
// Clock circuitry (250 ms cycle)
reg [???:0] cnt;
localparam CntMax = ???;

always @(posedge Clock)
        cnt <= (cnt == CntMax) ? 0 : cnt + 1;
```

Create a wire for holding the condition for executing the Instruction Cycle.

```
// Synchronise CPU operations to when cnt == 0
wire go = !Reset && (cnt == 0);
```

The Instruction Cycle then looks like this.

```
// Instruction Cycle - Instruction Cycle Block
always @(posedge Clock) begin
        // Process Instruction
        if (go) IP <= IP + 8'b1;
        // Process Reset
        if (Reset) IP <= 8'b0;
end
```

By putting the “if (go)” statement first, it is more likely to be used as the Clock Enable pin for the majority of CPU registers, once the rest of the design is implemented.

Question

Why will Quartus complain if the Instruction Cycle is split over two “always” blocks, with the first having “if (go)” and the second having “if (Reset)”?

Step 2 of Stage 5

We need to create the Program Memory, and place a program in there for the CPU to execute.

We have choices: synchronous or asynchronous memory, stored inside the FPGA or in external memory attached to the FPGA. We will take the simplest approach.

```
module AsyncROM(input [7:0] addr, output reg [34:0] data);
        always @(addr)
                case (addr)
                        0:   data = {`MOV, `PUR, `NUM, 8'd 1,  `REG, `DOUT, `N8};
                        1:   data = {`MOV, `PUR, `NUM, 8'd 3,  `REG, `DOUT, `N8};
                        2:   data = {`MOV, `PUR, `NUM, 8'd 5,  `REG, `DOUT, `N8};
                        3:   data = {`MOV, `PUR, `NUM, 8'd 7,  `REG, `DOUT, `N8};
                        4:   data = {`MOV, `PUR, `NUM, 8'd 9,  `REG, `DOUT, `N8};
                        5:   data = {`MOV, `PUR, `NUM, 8'd 11, `REG, `DOUT, `N8};
                        6:   data = {`MOV, `PUR, `NUM, 8'd 13, `REG, `DOUT, `N8};
                        7:   data = {`MOV, `PUR, `NUM, 8'd 15, `REG, `DOUT, `N8};
                        8:   data = {`MOV, `PUR, `NUM, 8'd 17, `REG, `DOUT, `N8};
                        9:   data = {`MOV, `PUR, `NUM, 8'd 19, `REG, `DOUT, `N8};
                        default: data = 35'b0; // Default instruction is a NOP
                endcase
endmodule
```

Place the above in ROM.v, and in your CPU module, add the following code.

```
// Program Memory
wire [34:0] instruction;
AsyncROM Pmem(IP, instruction);
```

Test

Temporarily include the following lines in your CPU module.

```
initial Dval = 1;
always @(*)
        Dout = instruction[25 -:8];
```

Then when you run your computer, the display will show 1,3, 5, 7, 9, 11, 13, 15, 17, 19 then remain on 0 until the IP reaches 00 again.

Questions

- How does Pmem(IP, instruction) work?
- What does the test code do?
- Why, after showing 19, does the display remain on 0 until the IP reaches 00 again?
- Are letters or numbers being stored into the program memory?

Step 3 of Stage 5

Add the following code to store the CPU Registers. Fix up any compiler errors.

```
// Registers
reg [7:0] Reg [0:31];

// Use these to Read the Special Registers
wire [7:0] Rgout = Reg[29];
wire [7:0] Rdout = Reg[30];
wire [7:0] Rflag = Reg[31];

// Use these to Write to the Flags and Din Registers
`define RFLAG Reg[31]
`define RDINP Reg[28]

// Connect certain registers to the external world
assign Dout = Rdout;
assign GPO  = Rgout[5:0];

// TO DO: Change Later
initial Dval = 1;
```

Here is a very simple instruction cycle.

```
// Instruction Cycle
wire [3:0] cmd_grp = instruction[34:31];
wire [2:0] cmd = instruction[30:28];
wire [1:0] arg1_typ = instruction[27:26];
wire [7:0] arg1 = instruction[25:18];
wire [1:0] arg2_typ = instruction[17:16];
wire [7:0] arg2 = instruction[15:8];
wire [7:0] addr = instruction[7:0];

always @(posedge Clock) begin
    if (go) begin
        IP <= IP + 8'b1;  // Default action is to increment IP
        case (cmd_grp)
            `MOV:
                Reg[arg2] <= arg1;
    // For now, we just assumed a PUR move, with arg1 a number and arg2 a register!
        endcase
    end
    // Place reset code here…
end
```

Test

This should display 1,3,5 etc when you run the program, due to the instructions loaded into memory in Step 2. Add several more instructions, to get some LEDs to light up. (Move a number into the `GOUT register.) You have built a very simple computer!

Stage 6

Implement the Unconditional Jump instruction, and test it out by adding a JMP to your Program Memory so that your program loops after ten or so instructions.

Note that by writing IP <= IP+1; before the case (cmd_grp) statement, you can change IP from within the case statement, and that change will take effect: according to the Verilog standard, the last assignment to a variable from within the **same** block is the one that will take precedence.

Turbo Switch

You can also implement the Turbo feature now. Add the following inside the CPU module.

```
wire turbo_safe;
Synchroniser tbo(Clock, Turbo, turbo_safe);
```

Then make a small change to the definition of go:

```
wire go = !Reset && ((cnt == 0) || turbo_safe);
```

The “safe” suffix reminds us that we have synchronised the signal.

Note

Have a look at the RTL for your CPU. This will be the last stage where the RTL is relatively straightforward to take in at a glance. After the next stage, even though only a few more lines of Verilog are added, there will be hundreds of wires everywhere!

Questions

- How does the Turbo feature work?
- What does it mean to have synchronised the turbo signal?
- What can go wrong if we had used Turbo instead of turbo_safe?

Stage 7

The full repertoire of the MOV Command Group will now be implemented. First create two Verilog functions to help us: get_number and get_location.

```
function [7:0] get_number;
        input [1:0] arg_type;
        input [7:0] arg;
        begin
                case (arg_type)
                        `REG: get_number = Reg[arg[5:0]];
                        `IND: get_number = Reg[ Reg[arg[5:0]][5:0] ];
                        default: get_number = arg;
                endcase
        end
endfunction

function [5:0] get_location;
        input [1:0] arg_type;
        input [7:0] arg;
        begin
                case (arg_type)
                        `REG: get_location = arg[5:0];
                        `IND: get_location = Reg[arg[5:0]][5:0];
                        default: get_location = 0;
                endcase
        end
endfunction
```

Then the appropriate part of the case statement can be changed to the following. Outside of the always statement you will also need to declare reg [7:0] cnum;. It will not be synthesised; it just makes the Verilog code easier to read. Since we are treating "cnum" as a temporary variable that does not get synthesised, we use "=" to assign values to it.

```
`MOV: begin
        cnum = get_number(arg1_typ, arg1);
        case (cmd)
                `SHL: begin
                        `RFLAG[`SHFT] <= cnum[7];
                        cnum = {cnum[6:0], 1'b0};
                end
                `SHR: begin
                        `RFLAG[`SHFT] <= cnum[0];
                        cnum = {1'b0, cnum[7:1]};
                end
        endcase
        Reg[ get_location(arg2_typ, arg2) ] <= cnum;
end
```

Note that very few lines of code were needed to implement a very powerful set of MOV instructions.

Test

Make sure your original program still executes correctly. Then change some of the instructions to use the new shifting ability of MOV commands, and check that it works.

Questions

- What do get_number and get_location do?
    - How do they relate to the CPU Instruction Set?
- What do the instructions `MOV `SHL and `MOV `SHR do exactly?
    - Why is an assignment made to the Flag Register?
- How is it possible that “cnum” is not synthesised? What does the synthesiser do instead?
- Look at the RTL. Why, with only a few lines of code, is there now a bird’s nest of wires?

Stage 8

Using get_number and get_location, it is not difficult to implement the ACC Command Group. It is expedient to introduce several more temporary variables that do not get synthesised. Here is a (big) hint: maybe you can have a cleaner version though.

```
`ACC: begin
        cnum = get_number(arg2_typ, arg2);
        cloc = get_location(arg1_typ, arg1);
        case (cmd)
                `UAD:   word = Reg[ cloc ] + cnum;
                `SAD:   s_word = $signed( Reg[ cloc ] ) + $signed( cnum );
                `UMT:   word = …; // Fill this in
                `SMT:   s_word = …; // Fill this in
                `AND:   cnum = Reg[ cloc ] & cnum;
                `OR:    cnum = …; // Fill this in
                `XOR:   cnum = …; // Fill this in
        endcase
        if (cmd[2] == 0)
                if (cmd[0] == 0) begin // Unsigned addition or multiplication
                        cnum = word[7:0];
                        `RFLAG[`OFLW] <= (word > 255);
                end
                else begin // Signed addition or multiplication
                        cnum = s_word[7:0];
                        `RFLAG[`OFLW] <= (s_word > 127 || s_word < -128);
                end
        Reg[ cloc ] <= …; // Fill this in
end
```

Test

Test out your computer with the following program.

```
case (addr)
        0:   data = {`MOV, `PUR, `NUM, 8'd 0, `REG, `DOUT, `N8};
        4:   data = {`ACC, `UAD, `REG, `DOUT, `NUM, 8'd 15, `N8};
        8:  data = {`JMP, `UNC, `N10, `N10, 8'd 4};
        default: data = 35'b0;
endcase
```

Questions

- Describe exactly how each Instruction works.
- Why does using the addresses 0, 4 and 8 introduce a delay?
- When the displayed Dout changes on the board, is the IP 4 or 5? Why?

Stage 9

Implement in full the JMP Command Group. Here is a skeleton.

```
`JMP: begin
  case (cmd)
    `UNC:         cond = 1;
    `EQ:          cond = ( get_number(arg1_typ, arg1) == get_number(arg2_typ, arg2) );
    `ULT:         cond = …
    `SLT:         cond = …
    `ULE:         cond = …
    `SLE:         cond = …
    default:      cond = 0;
  endcase
  if (cond) IP <= addr;
end
```

Test

Test it out with the following program.

```
0:   data = {`MOV, `PUR, `NUM, 8'd 1, `REG, `DOUT, `N8};
4:   data = {`ACC, `SMT, `REG, `DOUT, `NUM, -8'd 2, `N8};
7:  data = {`JMP, `SLT, `REG, `DOUT, `NUM, 8'd64, 8'd 4};
10: data = {`MOV, `PUR, `NUM, 8'd 100, `REG, `DOUT, `N8};
13: data = {`ACC, `SAD, `REG, `DOUT, `NUM, -8'd 7, `N8};
16: data = {`JMP, `SLE, `NUM, 8'd 0, `REG, `DOUT, 8'd 13};
20: data = {`JMP, `UNC, `N10, `N10, 8'd 0};
default: data = 35'b0;
```

Questions

- The thirty-two registers of the CPU are all unsigned. Yet we allow signed comparisons. How does this work?
- Explain why the above test program produces what it does.

Stage 10

The ATC Command Group is designed for reading the status of the push buttons. Nevertheless, we allow it to be applied to any bit of the Flag Register. Its implementation is straightforward.

```
`ATC: begin
        if (`RFLAG[cmd]) IP <= addr;
        `RFLAG[cmd] <= 0;
end
```

Syntactic Sugar Functions

Notice the use of functions for syntactic sugar that have been introduced below. It is up to you whether you want to create more syntactic sugar functions, or whether you prefer to enter instructions in raw form. Of course, you can also mix and match.

```
module AsyncROM(input [7:0] addr, output reg [34:0] data);
        always @(addr)
                case (addr)
                        0:  data = set(`DOUT, 1);
                        4:  data = acc(`SMT, `DOUT, -2);
                        8:  data = atc(`OFLW, 16);
                        12: data = jmp(4);

                        16: data = set(`DOUT, 250);
                        20: data = acc(`UAD, `DOUT, 1);
                        24: data = atc(`OFLW, 8);
                        28: data = jmp(20);

                        default: data = 35'b0; // NOP

                        1, 5, 9, 13, 17, 21, 25: data = mov(`FLAG, `GOUT);
                endcase

        function [34:0] set;
                input [7:0] reg_num;
                input [7:0] value;
                set = {`MOV, `PUR, `NUM, value, `REG, reg_num, `N8};
        endfunction

        function [34:0] mov;
                input [7:0] src_reg;
                input [7:0] dst_reg;
                mov = {`MOV, `PUR, `REG, src_reg, `REG, dst_reg, `N8};
        endfunction

        function [34:0] jmp;
                input [7:0] addr;
                jmp = {`JMP, `UNC, `N10, `N10, addr};
        endfunction
        function [34:0] atc;
                input [2:0] bit;
                input [7:0] addr;
                atc = {`ATC, bit, `N10, `N10, addr};
        endfunction

        function [34:0] acc;
                input [2:0] op;
                input [7:0] reg_num;
                input [7:0] value;
                acc = {`ACC, op, `REG, reg_num, `NUM, value, `N8};
        endfunction
    endmodule
```

Also try the move-with-shift instruction by making the following changes to the above:

```
        4:  data = {`MOV, `SHL, `REG, `DOUT, `REG, `DOUT, `N8};
        8:  data = atc(`SHFT, 16);
```

Questions

- What is being stored into program memory now? Functions? Letters? Or binary numbers?
- What does the first program do, and why?
- What about the two lines of changes, related to move-with-shift? What should they do?
- Why might an atomic instruction, such as ATC, be important in a CPU?

Stage 11

There is a loose end to tidy up. Remove the line “initial Dval = 1;”, change Dval to a wire, and make the following changes/additions.

```
assign Dval = Rgout[`DVAL];

// Debugging assignments – you can change these to suit yourself
assign Debug[3] = Rflag[`SHFT];
assign Debug[2] = Rflag[`OFLW];
assign Debug[1] = Rflag[`SMPL];
assign Debug[0] = go;
```

Setting and Clearing Individual Bits

How can we set or clear a particular bit of a register without affecting the other bits of the register? Since we did not create an explicit instruction to do this, the trick is to use an OR operation to set the bit we want, and an AND operation to clear the bit we want. (You may have come across these tricks before if you have programmed microcontrollers such as the Arduino.)

In Verilog, “<<” can be used to shift to the left. Note that 2’b1 << 2 will produce 00 because the result will be constrained to be 2 bits. Note 8’b1 << 2 will produce 0000_0100. To set bit number 2 of Reg[n] we can therefore use the ACC OR instruction to perform Reg[n] = Reg[n] | (8’b1 << 2).

To clear bit number 2 of Reg[n] we can use “~” to take the bitwise complement of 8’b1 << 2, then perform an AND operation: Reg[n] = Reg[n] & ~(8’b1 << 2).

This motivates the creation of the following two helper functions.

```
function [34:0] set_bit;
        input [7:0] reg_num;
        input [2:0] bit;
        set_bit = {`ACC, `OR, `REG, reg_num, `NUM, 8'b1 << bit, `N8};
endfunction

function [34:0] clr_bit;
        input [7:0] reg_num;
        input [2:0] bit;
        clr_bit = {`ACC, `AND, `REG, reg_num, `NUM, ~(8'b1 << bit), `N8};
endfunction
```

Modify your program so that the display continues to work.

Questions

- What does the line “assign Dval = Rgout[`DVAL];” do?
    - Why do we want that line?
- What change did you have to make to your program?
- Does your computer have to evaluate 8’b1 << bit every time you want to set or clear a bit? Explain.

Milestone Achieved

Congratulations! You should now have a fully functional instruction set. While the input buttons and switches are not yet connected, everything else concerning the CPU has been completed.

Once you have had a rest, and have gone back to tidy up your code, you will find that it takes relatively little Verilog code to produce a relatively powerful[3] (yet nonetheless very simple) computer.

You are close to finishing Part 1.

Remark

There are a number of extensions that you could consider, should you own your own DE1 board and wish to take this further the next time you have some holidays. Adding more instructions is a natural candidate.

Currently, there is no ability to have subroutines. One way to implement a basic form of a subroutine, without requiring a stack, is to create a new JMP instruction that can jump to the address in a given register. The idea is to store the return address in a register then jump to the location of your subroutine. The subroutine will jump to the return address given in the register when it has finished.

[3] I grew up with some of the very first personal computers, including the Super 80, the ZX Spectrum and later a Microbee. They used the Z-80 microprocessor, clocked at something like 2 MHz. Memory was in the order of 16 kilobytes. I remember programming the Super 80 using machine code, by hand, editing memory locations one by one, placing hexadecimal numbers in there corresponding to instructions, pretty much like you are doing for this project now. For the Super 80 and ZX Spectrum, programs were stored on cassette tape. Later, there was great excitement when we got a floppy disk drive for the Microbee…

Stage 12

The buttons and switches need connecting to the CPU registers. First and foremost, they require synchronisation, for otherwise the internal state of the CPU could be put in jeopardy.

Step 1 of Stage 12

Create a wire [7:0] din_safe and assign it to the outputs of eight Synchroniser modules whose inputs are the respective bits of Din. (But wait! Didn’t we learn that we should not use single-bit synchronisers to synchronise a multi-bit value? That is why we need some way of knowing when the value is “correct”, and for this, we use a push button: the user moves the switches to the desired position, then presses the left-most push button. However, we don’t want our design to break if the user moves the switches while pressing the push button, therefore, paranoia dictates that we synchronise Din before using it. For a similar reason, we put the synchronisers *inside* the CPU module rather than relying on them being provided externally.)

Also create a wire [3:0] pb_safe and assign its bits to the outputs of four Synchroniser modules whose inputs are Sample, Btns[2], Btns[1] and Btns[0].

Step 2 of Stage 12

Create a module DetectFallingEdge (in AuxMod.v). As inputs, it should have a clock and a btn_sync. Here, btn_sync must already have been synchronised to the clock, and should also be debounced. (The push buttons on the DE1-SoC board are debounced in hardware, according to the manual.)

The module's output should be "1" whenever a falling edge on btn_sync is detected (do not use the negedge keyword!). Preicsely, whenever the previous value of btn_sync is "1" and the current value is "0", a "1" should be output.

Your design should use exactly one flip-flop.

Questions

- Why is a flip-flop required?
- Why do we not want to use two or more flip-flops?
- Why not use negedge?

Step 3 of Stage 12

Connect all four synchronised push buttons to instantiations of the DetectFallingEdge module in the CPU module as follows.

```
genvar i;
wire [3:0] pb_activated;
generate
        for(i=0; i<=3; i=i+1) begin :pb
                DetectFallingEdge dfe(Clock, pb_safe[i], pb_activated[i]);
        end
endgenerate
```

After the "if (go) begin … end" in the Instruction Cycle block, add the following code, and fix any compile errors. Note that the Reset code now clears the Flag Register.

```
if (Reset) begin
        IP <= 8'b0;
        `RFLAG <= 0;
end
else begin
        for(j=0; j<=3; j=j+1)
                if (pb_activated[j]) `RFLAG[j] <= 1;

        if (pb_activated[3]) `RDINP <= din_safe;
end
```

Test

```
case (addr)
        0: data = set(`FLAG, 128);
        1: data = mov(`FLAG, `GOUT);
        2: data = mov(`DINP, `DOUT);
        3: data = jmp(1);
        default: data = 35'b0;
endcase
```

Remember you can use the Reset switch to clear the LEDs.

Questions

- Why might we want the CPU to automatically clear the Flag Register after a Reset?
- What does the above Verilog code do?
- What does the test program do?
    - Why is 128 loaded into the Flag Register?
- What happens if the Program Memory contains an instruction to store a value into Register 28 (`RDINP)? Will the value stay there? For how long? Which takes precedence if both a move instruction and the hardware try to change `RDINP at the same time?
- Is it possible for `RDINP to change halfway through when the programmer tries to move `RDINP into another register? When Turbo mode is off?
- Why might we have chosen to use the falling edge rather than the rising edge, or in other words, why do we want to detect push button releases rather than push button presses?
- When you first started, did you think you would ever finish Part 1?