CE2003 Laboratory 5
uP16

As seen in Lab4, the uP16 is a very basic 5-stage pipelined processor supporting full word access. It has a 16-bit PC which increments by 1 (unless there is a jump or branch instruction). uP16 has 8 general purpose registers, R_0 to R_7 with R_0 hardwired to 0. The remaining registers are general purpose registers. If required, a stack pointer (SP) and return address from a function call could be implemented using R_6 and R_7, respectively.

The uP16 memory space consists of two separate memory arrays: an 18-bit wide instruction memory (IM), configured as a 1024x18-bit single port ROM, and a 16-bit wide data/stack memory (DM), configured as a 1024x16-bit single port RAM. Hence, all uP16 instructions are 18-bit words stored in the IM. In this version of the processor, there is only one instruction format (called R-format), as in table 1. The instruction set consists of a very basic set of instructions (as in table 2).

Table 1: The 18 bit instruction formats for the uP16 processor

Field	4 bits	3 bits	3 bits	8 bits
R-format	OpCode	Rd	Rs	Function/Immed

Table 2: The instruction set for the uP16 processor

Name	op	Funct/ Immed	Instruction	Effect(s)
nop	0x00	0x00	**nop**	Do nothing (instruction word = 0)
add	0x00	0x01	**add** *rd*, *rs*	RF[rd] = RF[rd] + RF[rs];
sub	0x00	0x02	**sub** *rd*, *rs*	RF[rd] = RF[rd] - RF[rs];
and	0x00	0x03	**and** *rd*, *rs*	RF[rd] = RF[rd] **and** RF[rs];
or	0x00	0x04	**or** *rd*, *rs*	RF[rd] = RF[rd] **or** RF[rs];
nand	0x00	0x05	**nand** *rd*, *rs*	RF[rd] = RF[rd] **nand** RF[rs];
nor	0x00	0x06	**nor** *rd*, *rs*	RF[rd] = RF[rd] **nor** RF[rs];
xor	0x00	0x07	**xor** *rd*, *rs*	RF[rd] = RF[rd] **xor** RF[rs];
not	0x00	0x08	**not** *rd*, *rs*	RF[rd] = **not** RF[rs];
mov	0x00	0x0A	**mov** *rd*, *rs*	RF[rd] = RF[rs];
lw	0x01	--	**lw** *rd*, *rs, immed*	RF[rd] <- memory(RF[rs]+immed);
sw	0x02	--	**sw** *rd*, *rs, immed*	RF[rd] -> memory(RF[rs]+immed);
lli	0x03	immed	**lli** *rd, rs, immed*	RF[rd] = sign_ext(immed); (*rs* is unused)
lui	0x04	immed	**lui** *rd, rs, immed*	RF[rd] = {immed, RF[rs][7:0]}
addi	0x05	immed	**addi** *rd*, *rs*, *immed*	RF[rd] = RF[rs] + sign_ext(immed);
	0x6-8			**Reserved**
beq	0x08	immed	**beq** *rd, rs, immed*	if (RF[rd] = = RF[rs]) pc += sign_ext(immed) + 1;
bne	0x09	immed	**bne** *rd, rs, immed*	if (RF[rd] != RF[rs]) pc += sign_ext(immed) + 1;
blt	0x0A	immed	**bltz** *rd, rs, immed*	if (RF[rd] < RF[rs]) pc += sign_ext(immed) + 1;
bgt	0x0B	immed	**bgtz** *rd, rs, immed*	if (RF[rd] > RF[rs]) pc += sign_ext(immed) + 1;
	0xC-F			**Reserved**

Pre-Lab

You should look through the Module 8: uP Case Study lecture notes relating to data forwarding. You may also wish to examine Figure 1 and look at the Lab5 uP16 source code to see what is different from that of Lab4.

Task1

Open a new project called Lab5, targeting the Artix-7 xc7a35tcpg236-1 FPGA. Extract just the Verilog source files from uP_src.zip (inside CE2003_Lab5.zip) to a directory (called *up16_src*) inside the project directory. Add all the *.v* source files (as design sources) to the project. Add the constraints file *uP16_top.xdc* to the project. DO NOT add any of the other files in CE2003_Lab5.zip at this point.

1. Open the *uP16_top.xdc* file and change the clock constraint to use a 7.5ns clock, as:

   create_clock -add -name sys_clk_pin -period 7.5 -waveform {0 3.75} [get_ports Clk]

2. Run Synthesis. You will notice several warnings. Examine the warnings and the Verilog code to confirm that these can all be ignored. Then open the Synthesised Design, select "Report Timing Summary" and check that **All user specified timing constraints are met**. This design has a 7.5ns clock (but clock periods a bit less than 7.5ns will fail). This means that we could run this design up to about 133MHz.

3. Add *uP16_top_tb.v* (as a simulation source) from CE2003_Lab5.zip, and then run a **post synthesis timing simulation** (select "Run Simulation" and then select "Post-Synthesis Timing Simulation"). Then add the register file (RF) to your simulation. Select uut -> T3 -> EX_1, then in the objects window find *\RegFile_reg[0] [15:0]* (sometimes RF0 may be missing as it is always zero and may be optimised away by the synthesiser. If so, move on to RF1). If RF0 is present, it should have a value of 0000 and is an array data type. Drag it to the simulation window. Then find *\RegFile_reg[1] [15:0]* through *\RegFile_reg[7] [15:0]* and drag each of them to the simulation window. Make sure that they all have a multi-bit array data type, otherwise you have probably got the wrong signals.

   Then maximise the waveform window. Ensure the run time is set to 1us. Restart and Re-run (till 1us). You may need to do this a couple of times before all waveforms are correctly populated. Adjust the waveform so that it displays from 100ns to 400ns. You may need to "float" the window, by selecting the small square beside the cross (close) button at the top right of the waveform window.

4. Open the file *Lab4_timing_sim_screenshot.jpg* (inside CE2003_Lab5.zip) and compare it to your Lab5 waveform. Verify that there is NO real difference. Close all the simulation and Implementation windows until you are back at the PROJECT Manager window. Close the *Lab4_timing_sim_screenshot.jpg* file.

5. Examine the uP16 Block Diagram (Figure 1). Note that it has an extra control module in the ID stage (called *DF Control*). *DF Control* drives a pair of extra multiplexers in the EX stage (called S4 and S5) which direct data from the EX/Mem and Mem/WB pipeline registers into the ALU.

   At this point you may want to refer to the Module 8: uP Case Study lecture notes on data forwarding.

6. Open *ID_stage.v* and examine the *DF_control* module. Note that it has a port list but none of the inputs are used (which explains some of the unconnected port warning messages in Task1.2). The two outputs (*Data1_muxCont* and *Data2_muxCont*) are both set to zero. This means that data forwarding is disabled, and the processor pipeline behaves just like the version of uP16 shown in Figure 1 of Lab4.

7. Implement data forwarding by converting the pseudocode example (shown in Figure 2) to Verilog. Place your Verilog code inside the DF_control module just below the comment: "// You will need to comment out the above two statements and implement data forwarding control here". The variable names in the DF_control module port list will need to be strictly adhered to in your code. Correct any errors.

8. Run synthesis. There should be fewer warnings as those relating to unconnected ports in DF_control should no longer appear. Then open the Synthesised Design, select "Report Timing Summary" and check to see if the timing constraints have been met. If they are not, change the clock timing constraint to:

   create_clock -add -name sys_clk_pin -period 7.6 -waveform {0 3.8} [get_ports Clk]

   and rerun synthesis. Check that **All user specified timing constraints are met**. This design has a 7.6ns clock with an operating frequency of 131MHz (slightly less than the 133MHz of the original version). Change the clock timing constraint back to the original 10ns version:

   create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports Clk]

   Rerun synthesis. Then perform a post synthesis timing simulation. Again, add the RF elements to the simulation. Restart and Run for 1us (do this a couple of times until all waveforms are correctly populated). Again, adjust the waveform so that it displays from 100ns to 400ns and open the file *Lab4_timing_sim_screenshot.jpg* and compare it to your Lab5 simulation waveform. Verify that there is a difference in all register file locations except RF4. For example, consider the code:

   ```
   PC= 0x0004   lli   r2, #2          (instruction 0D002)
       0x0005   lui   r4, r4, #f9     (instruction 124F9)
       0x0006   add   r2, r1          (instruction 01101)
   ```

   The **lli** instruction starts at the rising edge of the clock at 165ns, the PC (0004) appears in IF_currPC at about 172ns, the instruction (0D002) appears at the output of the IF/ID pipeline register (as IF_inst_d) at about 182ns and is written into RF2 at about 208ns. This is as expected. The next instruction (**lui**) also behaves as expected and writes into RF4 at about 218ns. The 3rd instruction (**add**) operates differently to

what happened in Lab4. In Lab 4, **add r2, r1** used the stale version of r2 (RF2 equal to 0x0000) because the **lli** instruction had not yet written to RF2. It instead added 0+1 with a result of 1 (written back to RF2 at about 228ns). The Lab5 version adds the value of RF2 which is still in the pipeline (0x0002) with the value in RF1 (0x0001) to give a result of 0x0003, which is written back to RF2 (in Lab5) at about 228ns.

Also consider the code segment:

```
PC= 0x0013    add   r1, r6          (instruction 00E01)
    0x0014    lw    r7, r1, #1      (instruction 07901)
```

In Lab4, RF1 was not updated by the **add** instruction before it was used by the **lw** instruction. As a result the **lw** instruction used RF1=0x0001 and added the offset #1 to give an address of 0x0002, which was where the previous sw instruction wrote to, so **lw** retrieved 0xFFFF and wrote it to RF7 at about 338ns. However, in the data forward example, the address generated is 0x0001+0xF976+#1, which points to a location which contains the power-up value of 0x0000, so there is no change to RF7 at 338ns.

9. Close all the simulation and Implementation windows until you are back at the PROJECT Manager window. Close the *Lab4_timing_sim_screenshot.jpg* file.
10. Open *uP16_assembler.xlsx* and use it to replace the instruction initialisation code in *instruct_mem.v* (you will need to replace .INIT_00 to .INIT_02 and .INITP_00). Save the file and run a Behavioural Simulation (Not a Post-Synthesis Simulation). Add the RF, then Restart and run the simulation (at least 2 or 3 times). Then, examine the code between PC=0x0018 and 0x001D in *uP16_assembler.xlsx*. The **lw** instruction (at PC=0x0019) should load the value -1 (0xFFFF) into RF7 at about 415ns. This value was stored into memory location 0x0002 using the **sw** instruction at PC=0x0010. The next instruction (*add r2, r7* at PC=0x001A) which is a dependant instruction, should add RF2 (0x0003) with RF7 (0xFFFF) giving a result of 0x0002 which should be stored back into RF2 at about 425ns. However, the result stored back is 0x0005 which is incorrect. Why? The answer is that with data forwarding, the instruction immediately following a dependant instruction will use the ALU result at the output of the EX/Mem pipeline register (Note that a subsequent (second) dependant instruction would use the WB_result after the Mem/WB pipeline register). Obviously, if we are using whatever random value is currently stored at the output of the EX/Mem pipeline register, it is NOT the correct value from the data memory and will be incorrect. Hence, where there is a dependant instruction immediately after a **lw** instruction, we should use a ***load delay slot*** (like a branch delay slot). This could be a single **nop** instruction or some other non-dependant instruction, as:

```
lw    r2, r1, #2      // RF2 <= memory(RF1+2);
nop                   // wasted load-delay slot
add   r7, r2          // RF7 <= RF7+RF2
```

11. Now look at the next instruction (add r1, r7 at PC=0x001B). This is also dependant on R7. It should add RF1 (0x0002) with RF7 (0xFFFF, which is still yet to be written to RF7) giving a result of 0x0001 (which should be stored back into RF1 at about 435ns). Note that RF1 has the correct result because the correct memory output data (WB_result) was fed back from the Mem/WB pipeline register. Hence, only a single load-delay slot is needed (as in the code above).
12. Now in *uP16_assembler.xlsx*, examine the code between PC=0x001F and 0x0023. In the waveform window, move the cursor to 435ns (the rising edge of the clock at the start of PC=0x001F). Note that the PC sequence is 1F, 20, 21, 22, 23, 01. This means that the **beq** instruction at PC=0x0020 and 0x0021 were NOT taken, but the beq instruction at PC=0x0022 was taken (remember that because of the branch being calculated in the ID stage, the subsequent instruction (the **nop**) is also executed. This is because a branch is evaluated based on the contents of the register file. That is, data forwarding does not work as it only forwards data to the EX stage (not the ID stage). Hence, if a branch is dependant on a previous instruction, there must be two instructions between the previous dependant instruction and the branch instruction to ensure data is written back to the register file (these could be two **nop** instructions).

Task2

1. Edit the assembler code as:

```
PC= 0x001F    lli   r4, #0
              nop
              nop
```

```
                    beq    r4, r0, #9
                    nop
                    beq    r4, r0, #DD
PC= 0x0025          nop
```

Then replace the .INIT_00 to .INIT_02 and .INITP_00 instruction initialisation codes in *instruct_mem.v*.

2. Optional: At this point you can again do a behavioural simulation to see what the PC count sequence is with the new code.
3. Now implement the uP16 design on the Basys3 board. Follow the same procedure as in Task 3 of Lab 4. That is, remove the uP16-top.xdc constraints file and add the uP16_BASYS3_top.xdc constraints file. Then add the three design files, uP16_BASYS3_top.v, seven_seg.v and clkgen.v, as design sources.
4. At this point, again modify seven_seg.v to reflect your bench number.
5. Synthesise the design. Verify that any warnings can be ignored. Then implement the design and generate the bitstream. Then turn on the power to the Basys3 FPGA board and open the Hardware Manager and program the device

Inform your lab supervisor once you reach this point.

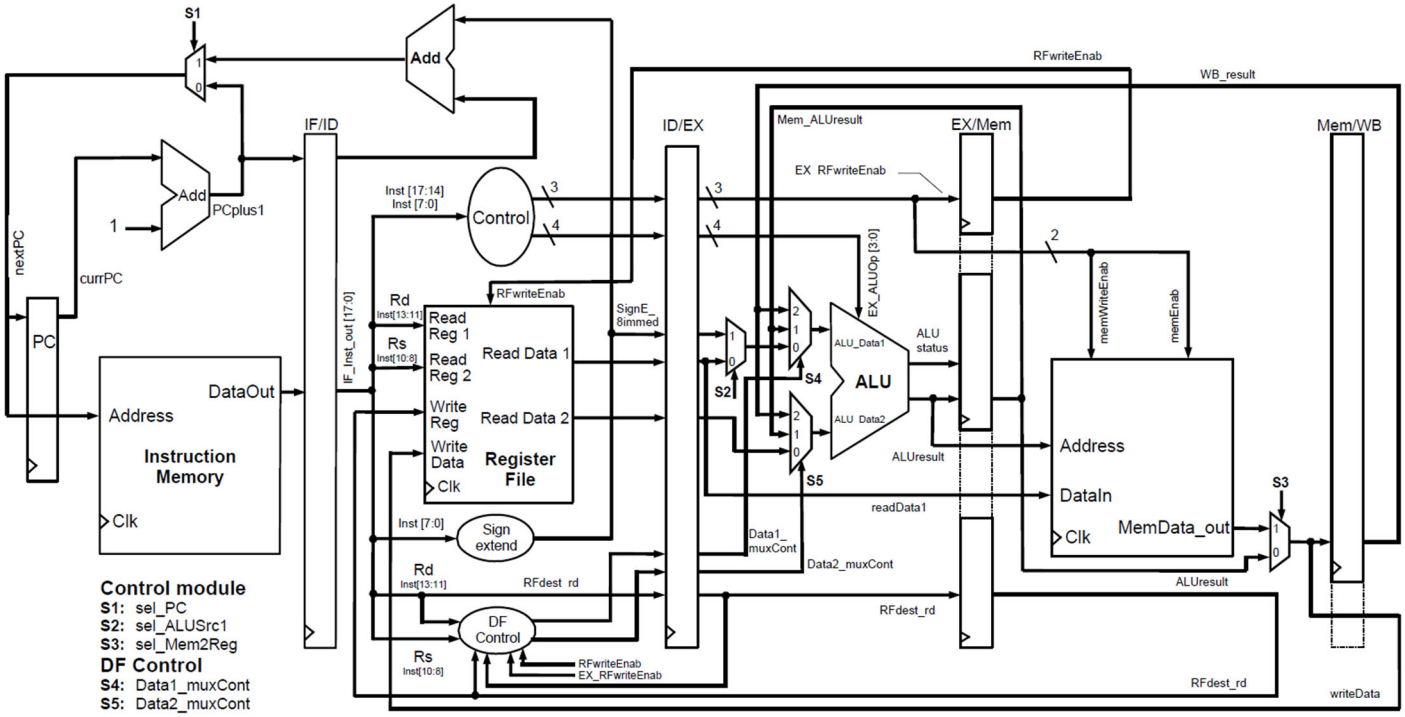

Figure 1: uP16 Block Diagram

Synthesiser warnings:

The following types of synthesiser warnings can be ignored

Task 1

[Synth 8-2507] parameter declaration becomes local in ??? with formal parameter declaration list

[Synth 8-689] width (32) of port connection 'DOADO' does not match port width (16) of module 'RAMB18E1'

[Synth 8-3331] design ??? has unconnected port ??? (**BUT**: you may want to individually check these)

[Constraints 18-5210] No constraints selected for write.

Additional warning for Task 2

[Synth 8-3917] design uP16_BASYS3_top has port anode_L[3] driven by constant 1

Forwarding (peusdocode)

If (*ID: rd* equals *EX: RFDest* AND *EX: RFwriteEnab)* set *S4cont* to ’01’
Else if (*ID: rd* equals *M: RFDest* AND *M: RFwriteEnab)* set *S4cont* to ’10’
Else set *S4cont* to ’00’ // Data comes from the RegFile module

If (*ID: rs* equals *EX: RFDest* AND *EX: RFwriteEnab)* set *S5cont* to ’01’
Else if (*ID: rs* equals *M: RFDest* AND *M: RFwriteEnab)* set *S5cont* to ’10’
Else set *S5cont* to ’00’ // Data comes from the RegFile module

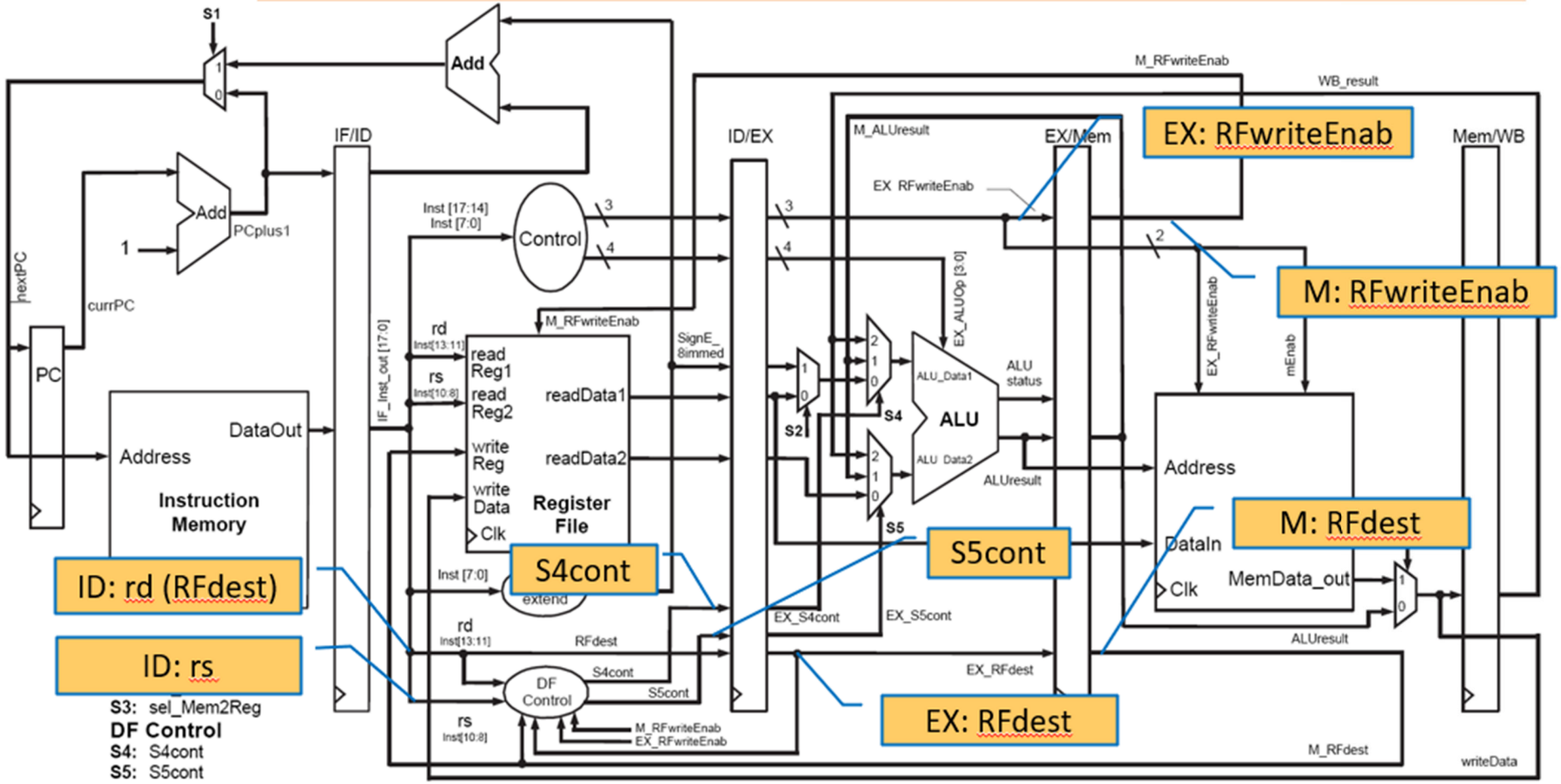

Figure 2: Slide from module 8: uP Case Study