# CE2003 Laboratory 4
# Pipelining and Datapaths

The uP16 is a very basic 5-stage pipelined processor. Only full word accesses are supported (that is, byte accesses are not supported). The program counter (PC) is 16 bits and indicates the next memory instruction to execute. As each instruction is executed, the PC increments by 1. On reset the PC is set to 0. There is no interrupt mechanism in the uP16 CPU. The block diagram of the uP16 processor is given on the last page.

**uP16 Register File:** The uP16 has 8 general purpose registers, $R_0$ to $R_7$. $R_0$ is hardwired to 0. Thus $R_0$ always reads as 0 and a write to $R_0$ has no effect. The remaining registers are general purpose registers. A stack pointer (SP) and return address from a function call could be implemented using $R_6$ and $R_7$, respectively.

**uP16 Memory:** 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. The memories are implemented as instantiated block RAM (BRAM) macros, rather than using the IP CORE Generator Wizard that you used in Lab 2. This is because the instantiated macro makes it easier to modify and re-initialise the contents of the Instruction ROM as well as being easier to view the contents in the simulator, compared to the Wizard implemented module.

**uP16 Instructions:** All uP16 instructions are 18-bit words stored in the IM. There is only one instruction format (called R-format) as in table 1. The instruction set consists of a very basic set of instructions (table 2). Note that there are no jump instructions and only signed arithmetic operations are supported. The instruction (the contents of the instruction memory), is the 5 hex digits in the *Code* field of *uP16_Assembler.xlsx*.

**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

Examine the *IF_stage.v*, *ID_stage.v*, *EX_stage.v* and *Mem_stage.v* Verilog source files (note from figure 1, there is no write back (**WB**) stage with write back occurring on the clock edge at the end of the **Mem** stage). In *ID_stage.v*, look at the control module implementation (particularly the instruction decoding and control signal generation). Study the implementation of the 16-bit sign extend unit (a single line immediately below the module instantiations).

## Task1

Open a new project called Lab4, targeting the Artix-7 xc7a35tcpg236-1 FPGA. Extract the Verilog source files from uP_src.zip (inside CE2003_Lab4.zip) to a directory (called *up16_src*) inside the project directory. Add all the source files (as design sources) to the project. DO NOT add any of the files in CE2003_Lab4.zip at this point.

1. Open *Instruction_mem.v* (a sub module under *IF_stage*) and examine the *I_mem* module. Note that it has a port list with just a clock, address and data out (the minimum signal requirement for a synchronous single port ROM). Also note the instantiated **BRAM_SINGLE_MACRO** macro with instance name **BRAM_SINGLE_MACRO_inst0** (below the .INIT initialisation information) and port identifiers DO to WE. Examine the signal mapping to the *I_mem* port list.
2. Examine the .INIT statements in the parameter section of the **RAMB16_S18** instance. This initialises the BRAM memory contents at startup (needed as it is a ROM). There are 2 different initialisations. The .INIT_00 to .INIT_3F statements initialise the 1024x16 data section of the memory array, while .INITP_00 to .INITP_07 initialises the 1024x2-bit parity section. We are not using the parity bits as parity, and instead we are using them as extra memory bits to give a 1024x18-bit memory. The important thing is that the two blocks need to be initialised separately. Notice that currently everything is set to zero. Next, open the uP16 assembler (*uP16_Assembler.xlsx)* and examine the blue shaded section. This allows you to enter instructions which are converted to assembly and binary, in the *Assembly Code* and *Code* columns, respectively. Replace the second *nop* (at row 4/memory address 0x02) with "**lli r3 ff**", like the instruction immediately above. Notice the change in the *Assembly Code* and *Code* columns. Examine the *RAM Data init* and *RAM Parity init* columns. They contain the .INIT and .INITP expressions. Replace the first three .INIT statements in *Instruction_mem.v* with the three non-zero .INIT statements from the *RAM Data init* column in *uP16_Assembler.xlsx*. Also replace the first .INITP statement with the first .INITP statement from the *RAM Parity init* column in *uP16_Assembler.xlsx* (you may need to scroll down the page a bit to find the .INITP_00 statement). Save all files.
3. Add the constraints file (*uP16_top.xdc*) to the design.
4. Synthesise the design. Examine any errors or warnings There are several warnings, but they all should be able to be ignored. Open the Synthesised Design and check that **All user specified timing constraints are met**. This design has a 10ns clock (specified at the top of the *uP16_top.xdc* file). Note (but there is no need to do this step) that a constraint with a 7.5ns clock will also work, but clock periods less than that will fail. This means that we could run this design up to about 133MHz.
5. Add the testbench (*up16_top_tb.v*) as a simulation source to your design. Open the testbench. Note that it just has a reset (*Rst*) and a 10ns clock (*Clk*). Note, no other stimuli are needed as the instructions are pre-initialised in the instruction memory (above).
6. Simulate the design. **Check that the default simulation radix is hexadecimal** (Select the settings icon at the above-right of the black waveform window). Zoom out till you can just see the digits in IF_inst_d. Observe that after *Rst* is de-asserted the program counter (PC), *IF_currPC*, increments from 0000 each clock cycle. Also note that the processor instruction, *IF_inst_d*, which is read from the instruction memory (addressed by the PC) is one clock cycle later, as it is actually from the IF/ID pipeline register (see figure 1).
7. Add the register file (RF), *RegFile [0:7,15:0]*, and Data memory Write_Enab, Enable, Add_In, Data_in, Data_out simulation objects to the simulation. To do this open the *uP16_top_tb* instance in the "Scope" pane. Open *uut*, and then the *T3* instance. Select the EX_1 sub-instance (the various instances match the module instantiation instances in uP_top.v) until you have the *RegFile* element. Then drag the simulation object (from the 'Objects" pane) to the simulation "Name" pane. Repeat for the *T7* instance, selecting the DM_1 sub-instance, and then drag the required 5 instances. In the waveform window (in the "Name" pane), expand *RegFile*. **Save the simulation window to Lab4 directory and add to project**.
8. Set the run time for 1us and the Restart and Re-run the simulation for the time specified (you may need to do this twice if there is missing data). Maximise this window and zoom as necessary.

9. Click on the waveform at the second rising edge of *Clk* AFTER *Rst* goes low (at 135ns). The yellow curser should move to there. Observe that the first lli instruction (lli r4, #71) in the excel file occurs at PC=0x0001. This instruction should put 0x71 into *RegFile*[4]. Note that it is written into the RF at the start of the 5th cycle after PC=0x0001. The next instruction is the one you added (lli r3 ff). Note that *RegFile*[3] changes one cycle later but has a value 0xFFFF. This is because lli sign extends the 8-bit value.
10. Next scroll (and click) at 245ns. PC=0x000F is the first sw instruction (sw r4, r1, #0), which stores the contents of RF4 (0xF971) into the memory location pointed to by the sum of RF1 (0x0001) and the offset (0x000). Observe that the two memory enables, address and data in signals are the expected values and are active 2 cycles later. Two cycles delay (rather than the expected 3 cycles) is correct as an examination of Figure 1 shows that these signals are generated in the EX stage and bypass the EX_Mem pipeline register. But the memory is synchronous (clocked) and the write does occur on the next clock cycle (as indicated by the data on the *Data_out* signal occurring one clock cycle later).
11. Next scroll (and click) at 275ns. PC=0x0012 is the first lw instruction (lw r6, r1, #0), which stores the contents of the previously written memory location to RF6. Observe that the memory enable and address are as expected and are active 2 cycles later (*Data_in* is not active and just contains some random data). The *Data_out* signal has valid data at the start of the 4th cycle after PC=0x0012, while RF6 is valid at the start of the 5th cycle after PC=0x0012, as expected.
12. Lastly, scroll across to PC=0x0015 (at 305ns). If you examine the excel file you will see that there are two consecutive add instructions (add r3, r1 followed by add r5, r3). In the simulation window, note that RF3=0x0002 and RF1=0x0001, so the sum should result in 0x0003 being written to RF3 at the start of the 5th cycle after the curser. Observe that this occurs as expected. However, the 2nd add (which occurs 1 cycle later) should add RF5=0xFFFF (or -1) and RF3 (which should be 3 from the previous add), giving a result of 2. But note that RF5 (which does change 5 cycles later) is actually 0x0001. This is because data forwarding has NOT been implemented in this version of the uP16 processor. The value of RF3 which is actually used is the old stale value (of 0x0002) as the new value (0x0003) has not yet been written back to the register file. Hence the result is -1+2 = 1, which is seen after 355ns. Note that there are earlier errors due to no data forwarding. For example, look at PC=12 and PC=13. **Save the simulation window.**

## Task2

If not already done. expand you Vivado window to full screen, then maximise your simulation window. Scroll the simulation window so that it starts just before reset goes low. You should see from 115ns to somewhere near 300ns (or more). Use a snipping tool to capture the waveform window from 115ns to 300ns. Save this for later. **Now close the Simulation window** (by clicking on the X on the RHS of the blue SIMULATION bar).

Now run a post synthesis timing simulation (note if your synthesis is out of date you may be asked to re-synthesise). To do this, select “Run Simulation” and then select “Post-Synthesis Timing Simulation”.

1. As before add the register file (RF) to your simulation. Select uut -> T3 -> EX_1, then in the objects window find *RegFile_reg[0] [15:0]* (it should have a value of 0000 and is an array data type). Drag this 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 an array data type, otherwise you have probably got the wrong signals. Then maximise the waveform window. Ensure the run time is set to 1us (running for 100us takes too long). Restart and Re-run (till 1us). You may need to do this twice.
2. Click on 135ns (the yellow cursor should move to there). This is the clock cycle at the start of the lli instruction (lli r4, #71) you examined in Task 1. Note that the point where PC=0x0001 now occurs 2/3rds of a clock cycle later. Now click at 175ns (the start of the 5th cycle) and observe that 0x0071 is written back into the register file during the 5th cycle (about 1/3rd of a clock cycle later, no longer at the start).
3. Also observe that the memory is operating correctly. Remember we used a sw instruction to store the value in *RegFile*[4] to memory and then subsequently used a lw instruction to write back to *RegFile*[6]. These instructions occurred at PC=0x000F (245ns) and PC=0x0012 (275ns), respectively. Verify that the contents of *RegFile*[4] at 245ns is written back to *RegFile*[6] at about 320ns.
4. Compare the timing simulation with the saved behavioural simulation. Is there any difference (apart from the delay)? The values should be the same, but instead of changing at the rising edge of the clock, the signals change after the clock (about 2/3rd of a clock cycle later for the output signals and about 1/3rd of a clock cycle later for the register file values). This is because this simulation takes the propagation delays through the circuit into account.
5. You can now close the simulation window. Also close the Synthesised Design window.

## Task3

Now implement the uP16 design on the Basys3 board. The Basys3 board has been configured to display the lower 8 bits of the PC on the seven segment display. The system clock has also been slowed down so that you can see the sequence of instructions as they are being executed. Additionally, the uP16 status is output to the board LEDs.

1. Remove the *uP16-top.xdc* constraints file and add the *uP16_BASYS3_top.xdc* constraints file.
2. Add the three design files, *uP16_BASYS3_top.v*, *seven_seg.v* and *clkgen.v*, as design sources.
3. At this point, again modify *seven_seg.v* to reflect your bench number.
4. Synthesise the design. Verify that any warnings can be ignored.
5. Implement the design and then generate the bitstream. Then turn on the power to the Basys3 FPGA board and open the Hardware Manager and program the device
6. Verify that 7-segment display matches the PC sequence from the saved behavioural simulation. It is:

   **00 to 0B, jump to 0F, continue sequentially to 1B, jump to 21, 22, 23, then jump back to 01 and repeat.**

**Inform your lab supervisor once you reach this point.**

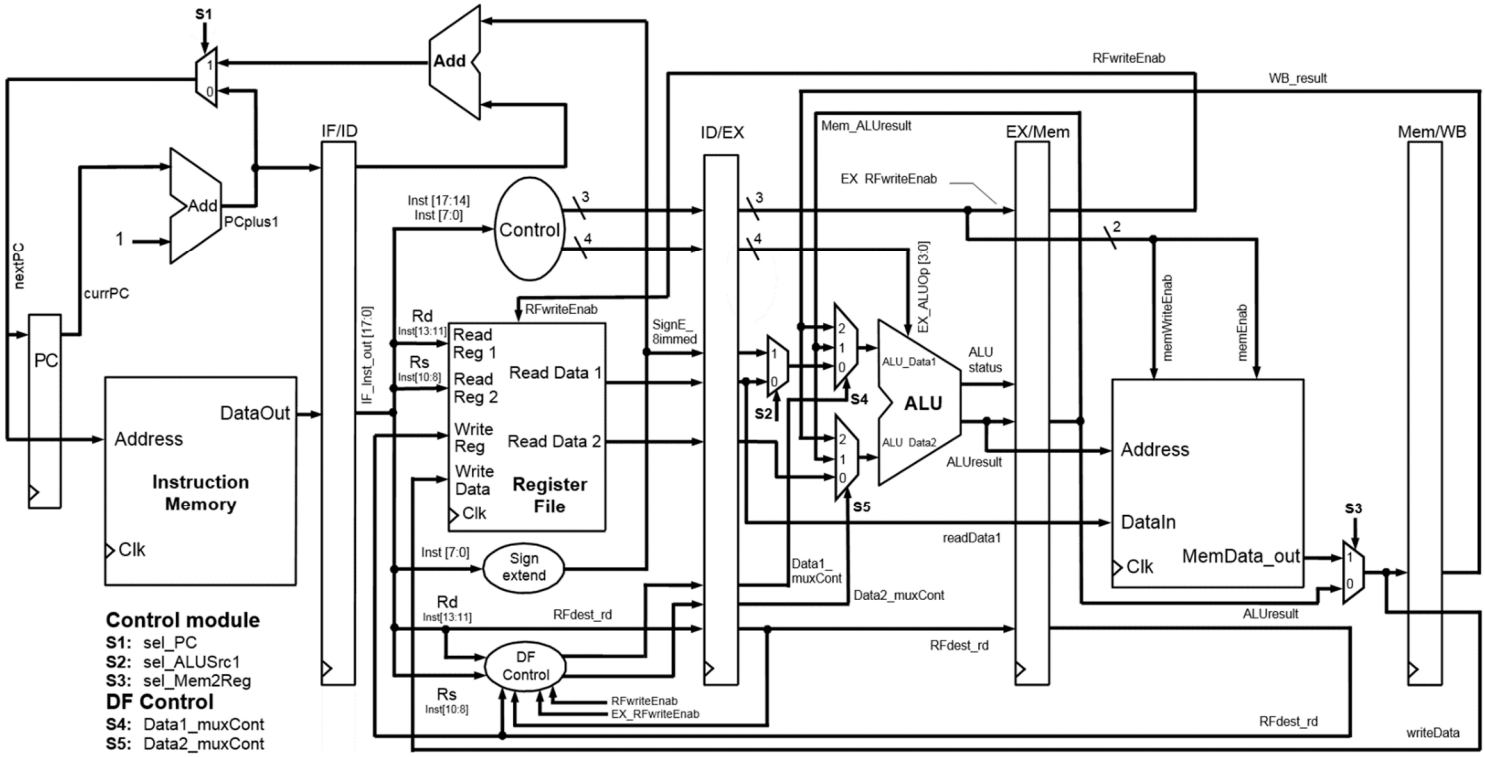


Figure 1: uP16 Block Diagram

## 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 Mem_stage has unconnected port ???

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

Additional warning for Task 3

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