Chapter 3

Lab 3 - Program Flow

3.1 Introduction

In this lab you will learn more about what special considerations are needed when designing logic for program flow instructions. You are allotted two lab occasions (eight hours) to finish this lab.

3.2 General description

After finishing lab 2 all building blocks of the processor are completed. It is now time to put all the parts together. In a pipelined processor this is not as easy as it first might seem. The parts must not only fit together in the topological aspect of connecting the right wire to the right input. The parts must also fit together in time. In Senior this task is complicated by the fact that there are three different pipeline depths in the processor, the pipeline depth depends on the instruction being executed. A pipeline will also make all kinds of program flow instructions troublesome as will be shown later in this exercise.

During all execution, instructions are fetched from the program memory (PM) using the program counter (PC) as the address. An instruction decides the values of all the control signals in the processor. An instruction usually has fewer bits than there are control signals in a processor, thus the instruction must be converted into its control signals before it is of any use. And that is the task of the instruction decoder. The instruction decoder, situated in pipeline stage P2 in the processor according to Figure 3.1, has two sub modules; `id_decode_logic` and `id_pipeline_logic`. Verify this by open `instruction_decoder.v`. Your first task is to briefly describe what the two modules do. It is important that you understand how the pipeline in Figure 3.1 works in order to complete this lab.

In lab 2 you have been running the entire processor, but not necessarily had to look into it in any detail. In this lab it is advised that you familiarize yourself a bit more with the processor and the system supporting it, see section 0.8.

3.3 Task 1: Instruction Decoding Hardware

Describe what the module `id_pipeline_logic` and `id_decode_logic` do. Hint: *Their functionality is not far from their names.*.

Jumps, calls and returns are essentially the same operation and will be referred to as just jumps if the distinction is not important.

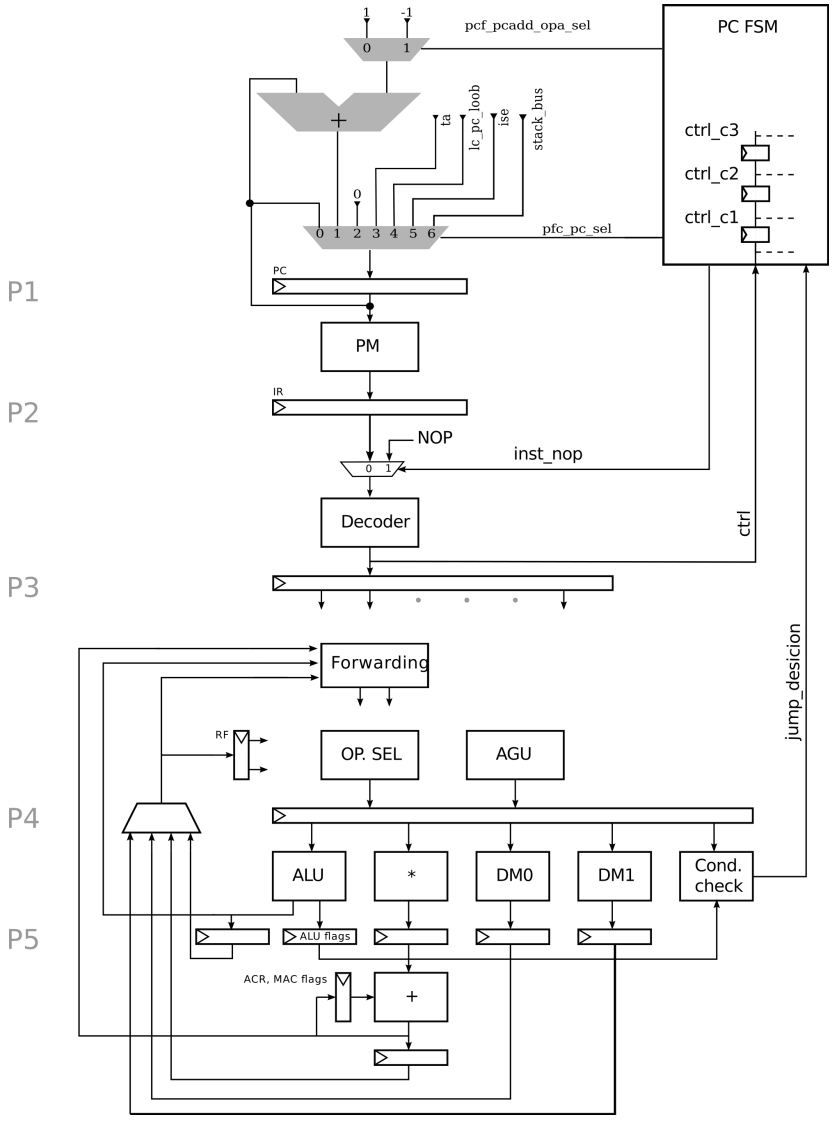


Figure 3.1: The normal pipeline, including PFC details.

3.3.1 Task 1: Summary

Answer the following:

1. Describe what the modules `id_pipeline_logic` and `id_decode_logic` do.

3.4 Understanding Delay Slots

Due to the pipeline the actual PC update in a jump can not and will not happen immediately. The previous instruction must have reached and set the ALU flags in pipeline stage P4 to make conditional jumps work correctly. MAC flags will be set one cycle later and if using them to jump on the programmer must ensure they are set correctly. Waiting for the correct flags to be set means that a number of instructions after the jump instructions will be fetched and executed. The question is then what to do with them. There are two extreme options. The first extreme option is to let the processor execute them as they are fetched and it is the programmers responsibility to make sure that the instructions after the jump instruction are useful. Instructions after the jump are said to reside in the jumps delay slots. The safe way to handle delay slots is to insert only `NOP`:s in all slots. The other extreme option is to let the processor deal with this and always insert `NOP` instructions in place of the fetched instructions. In Senior you can chose what you want to do with the `ds` directive. Thus when using `ds1` with a `jump` instruction the instruction following the `jump` will execute but the other two instructions will be forced to `NOP`:s by the processor. This is the reason for the `pfc_inst_nop_o` signals from the PC FSM. If it is set high, the decoder will see a `NOP` instruction instead of the instruction fetched from PM.

P0 **PC+**	**P1** **PC(instr)**	**P2**	**P3**	**P4**
1	0(A)	—	—	—
2	1(B)	A	—	—
3	2(JMP 0)	B	A	—
4	3(C)	JMP 0	B	A
5	4(D)	C	JMP 0	B
0	5(E)	D	**C**	JMP 0
1	0(A)	E	**D**	**C**
2	1(B)	A	**E**	**D**
3	2(JMP 0)	B	A	**E**
4	3(C)	JMP 0	B	A
5	4(D)	C	JMP 0	B
and so on				

Table 3.1: Pipeline table for jump instruction.

3.5 Pipeline Table

To fully understand what is going on in the pipeline of a processor it is useful do draw a pipeline table as shown in table 3.1. This example shows how a jump instruction is executed. In the P0 column the next PC value is listed, in P1 the actual PC value together with the instruction at that address in the program memory is listed, in P2 – P4 the instruction is simply propagated through the pipeline. Here it is clearly seen how, in P3 and P4, three extra instructions are executed (C,D, and E), that is if nothing is done

about it, i.e. inserting `NOP`s as described earlier. If you find it troublesome completing the PC FSM, it is recommended that you draw a pipeline table for all program flow instructions, i.e. `JUMP`, `CALL`, `RET`.

3.6 Task 2: Pipeline Table of a Small Program

In listing 3.1 a small program is defined. Your task is to fill out a pipeline table as described in section 3.5. Detailing how the instructions progress in the pipeline. You can refer to the instructions as the line number they are defined on in listing 3.1. Set PC to zero for the first instruction (on line 2). When doing this exercise look at the pipeline of the processor, shown in Figure 3.1, and think about what instructions are in each pipeline stage.

Also answer the following questions:

1. Which instructions get executed and which instructions get flushed?
2. What are the differences between a return and any other jump instruction? Hint: *The value of the return PC must be fetched from the stack.*

Listing 3.1: A small program(`lab3_ex.asm`)

```
1           .code
2           set r1,0x0001
3           set r2,0x0001
4           sub r0,r1,r2
5           jump.eq start
6           add r13,r1,r2
7   start
8           add r1,r1,r2
9           jump ds2 l1
10          add r1,r1,r2
11          add r2,r1,r2
12          add r3,r1,r2
13  l1
14          call ds3 f1
15          add r4,r1,r2
16          add r5,r1,r2
17          add r6,r1,r2
18          move r7,r2
19
20          out 0x12,r0
21
22  f1
23          move r8,r1
24          ret ds1
25          add r9,r1,r2
26          add r10,r1,r2
27          add r11,r1,r2
28          add r12,r1,r2
```

3.6.1 Task 2: Summary

Create a pipeline table of the program in listing 3.1 and answer the following questions:

1. Which instructions get executed and which instructions get flushed?
2. What are the differences between a return and any other jump instruction? Hint: *The value of the return PC must be fetched from the stack.*

3.7 Task 3: Program Counting

Before any instruction decoding or pipelining can take place the instruction to use must be fetched. This is done by taking the value of the program counter as an address into the program memory. In a processor with no flow control, the program counter logic is easy. Just increment it by one instruction each clock cycle. However in a useful processor some form of flow control must be implemented. Before solving this task, the program counter will only increment its value by one each clock cycle and executing any form of program flow instructions will result in undefined behavior.

3.7.1 The PC FSM

Your task is to complete the finite state machine responsible for selecting what the next value of the program counter shall be. Large parts of the FSM has been implemented in `pc_fsm.v(hd)`. Your task is to complete the FSM. You need to make sure that the FSM jumps to the correct state and also in each state sets the correct output signals.

As shown in Figure 3.1, the PC FSM takes inputs from the instruction decoder(the `ctrl` signal) and the condition checker(the `jump_decision` signal). It outputs two signals, `pc_add_opa_sel` and `pc_sel` to the program counter (next PC block). The program counter will update the PC value according to these two control signals from PC FSM.

In order to complete this task you must understand the architecture of the program counter (`program_counter.v`), shown in Figure 3.2. Worth noting about the program counter architecture is that two of the output signals from the PC FSM is used to select the next PC.

Control Signal Naming Convention

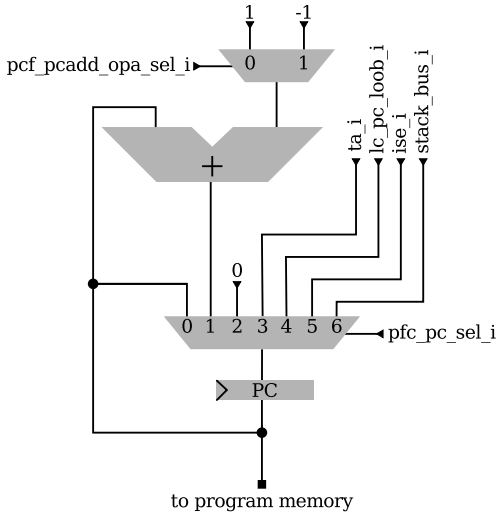
The control signals names in the modules is named as follows: `ctrl_`$\xi\Delta\gamma$. Where ξ is `_i` or `_c`μ, where μ is an integer representing how much the control signals has been delayed locally in the module. Δ is the control signal name delimiter, ` in Verilog or _ in VHDL. And $\gamma$ is the control signal name. For example from table 3.2, `ctrl_c2_PFC_RET` (VHDL), is the signal `ctrl_i_PFC_RET` delayed two clock ticks.

Further you have to understand how the PC FSM works. The PC FSM is of Mealy type, which means that both the next state and the output is a function of the current state and the input signals. The general architecture of a Mealy machine is shown in Figure 3.3.

The state transition graph for the PC FSM is shown in Figure 3.4. Worth noting are the five different paths going from S0. There are four different paths for jumps depending on how many delay slots are used. And finally there is one path for normal PC increment.

There are two subtasks in getting the PC FSM to work. One task is to get all the transitions right and the other task is to make sure that each state outputs the correct values. Its not necessary to solve one task before the other, it might be easier to solve them together, one instruction at a time. To solve both these subtasks you must understand the meaning of the in and output signals to/from the PC FSM, the signals are listed in table 3.2 and table 3.3.

a)



b)

signal	width	description
ise_i	16	interrupt address (not used)
lc_pc_loopb_i	16	start address of repeat loop (not used)
ta_i	16	target address of a jump or call
stack_bus_i	16	top of hardware stack, used for returns
pfc_pcadd_opa_sel_i	1	inc/dec PC
pfc_pc_sel_i	3	next PC mux selection

Figure 3.2: Next PC selection. a) Block diagram b) Signal descriptions

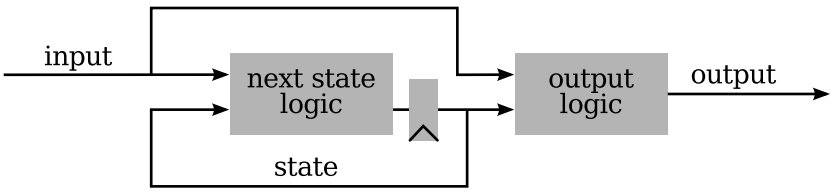


Figure 3.3: General architecture of a Mealy type FSM.

It is important to note that all modules involved in program counting get their control signals unpipelined from the instruction decoder. This in effect means that the loop controller and PC FSM gets their control signals in pipeline stage P2.

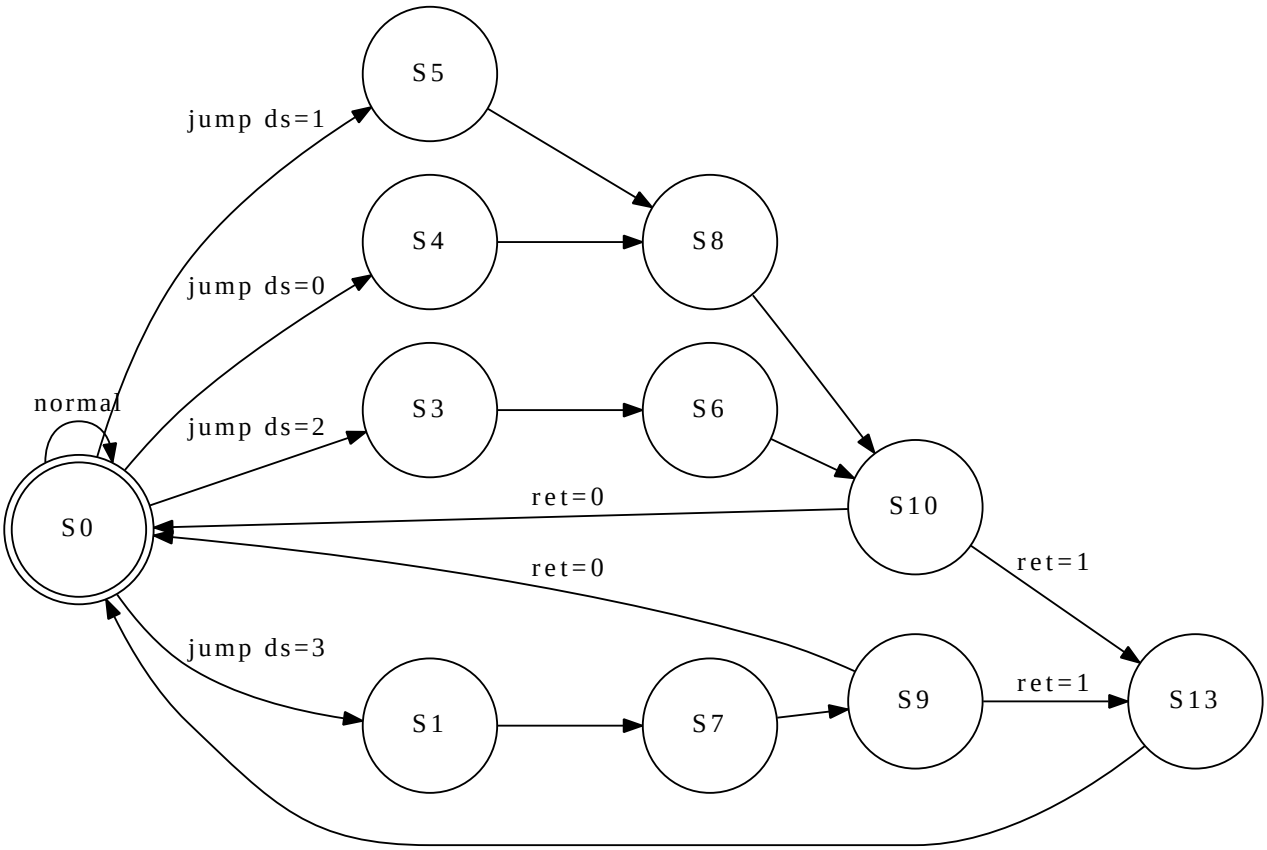


Figure 3.4: The PC FSM

PC FSM Transitions

As can be noted in your skeleton file S0 will only jump to itself. This is obviously not correct and you must select what state to jump to from S0.

Hint: *Look for the "What is the next state?" comment.*

PC FSM Output

All the output signals from the PC FSM are listed in table 3.3 and it is your task to make sure they are set correctly in each state.

3.7.2 Testing the PC FSM

There are a number of assembler files prepared for you to test the RTL code you have written, they are listed in table 3.4. It is suggested that you run the test programs in the order they are listed in table 3.4

signal	width	description
jump_decision_i	1	high if jump shall be taken.
lc_pfc_loope_i	16	end address of repeat loop (not used)
lc_pfc_loop_flag_i	1	high if loopn register is 0 (not used)
pc_addr_bus_i	16	current PC value
ctrl_i	6	control signals from the instruction decoder
ctrl_i_PFC_REPEAT_X	1	high if repeat of only x instructions (not used)
ctrl_i_PFC_JUMP	1	high if executing a jump instruction
ctrl_i_PFC_DELAY_SLOT	2	number of delay slots after a jump
ctrl_i_PFC_RET	1	high if executing a return instruction

Table 3.2: PC FSM input signals. The signal `ctrl_i` is divided into 4 fields.

signal	width	description
pfc_pc_add_opa_sel_o	1	what to add to PC (see Figure 3.2)
pfc_pc_sel_o	3	final PC selection (see Figure 3.2)
pfc_inst_nop_o	1	if high, inserts a NOP instruction
pfc_lc_loopn_sel_o	1	if high, decrement the loop counter register (not used)

Table 3.3: PC FSM output signals

since the later programs might assume that some program flow instructions are working. To run the test programs, just follow the procedure outlined in section 0.6.

File	Description
pfc_jump.asm	Tests `jump`.
pfc_cond_jump.asm	Tests conditional `jump`.
pfc_call.asm	Tests `call` and `ret`.

Table 3.4: PC FSM test programs

3.7.3 Debugging hints

If you encounter any bugs in your PC FSM it is much easier to debug them if you create your own test programs. (For example, by copying one of the test benches provided by us and removing everything but one test.)

Another fairly common issue in this lab is that you may create a combinational loop which is fairly hard to find. If your simulation time suddenly stops advancing you have probably managed to create a combinational loop. **Hint:** Look at Figure 3.1 and determine whether you are setting the control signal for the NOP mux correctly.

We have also noticed that missing signals in the sensitivity list in a process is a relatively common issue which can cause all sort of weird bugs.

3.7.4 Task 3: Summary

Complete the HDL code for the PC FSM as outlined above. Then test it using the test programs listed in table 3.4.

3.8 What to Answer

When you have completed all lab tasks, you should demonstrate your design, show the code you have written and be prepared to answer questions on how your design works.