Lab 3: Pipelined RISC-V core

1 Pipelined architecture overview

Unlike the single cycle core, where the instruction is read, processed, calculated, and stored within the same clock cycle, pipelined cores split this up int multiple sections. This has the benefits of reducing the critical path[1], allowing the processor to operate at a faster clock speed. In addition, the processor uses fully sequential memory, allowing compatibility with more efficient memory units.

Pipelines come with the drawback of more complexity such as pipeline hazards, branch prediction, forwarding, and synchronization. *Figure 2* details an implementation of a 5 stage pipelined RISC-V core. The stages are Fetch instruction, Decode, Execute, Memory, and Write back. Instructions flow through the core like an assembly line, and each stage has its specific function *Table 1*.

Stage	**Function**
Instruction Fetch (IF)	Reads from the Program Memory and returns the instruction as well as current program counter value.
Decode (DEC)	Takes in instruction, returns control signals as well as immediate for use in branching and further stages.
Execute (EXE)	Data is forwarded if necessary, and the result is calculated in the ALU.
Memory (MEM)	Responsible for writing and reading from the RAM.
Write back (WB)	Writes the result back to register memory.

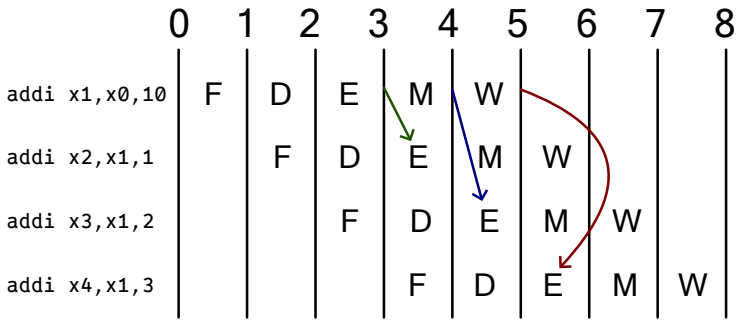

Figure 1: Forwarding illustration

There are cases where an instruction requires data that is written into the register memory by the previous instruction. However, an instruction can reach the execute stage before it has access to this data in a 5 stage pipeline. In order to avoid stalling, forwarding paths should be included to get this data while the instruction is in the execute, write, or write back phase.

Figure 1 shows an exaple with four addi instructions. The latter 3 all reference x1, has not yet been written to in their decode phase. Forwarding check, during the execute phase, whether one of its source registers is to be written to by data in EXE, MEM, and WB registers (in that order). It then extracts this for use as its true source, instead outdated data inside regMem.

[1]The critical path is the longest combinational chain of logic and it bottlenecks clock speed.

1.1 RISC-V Data Path

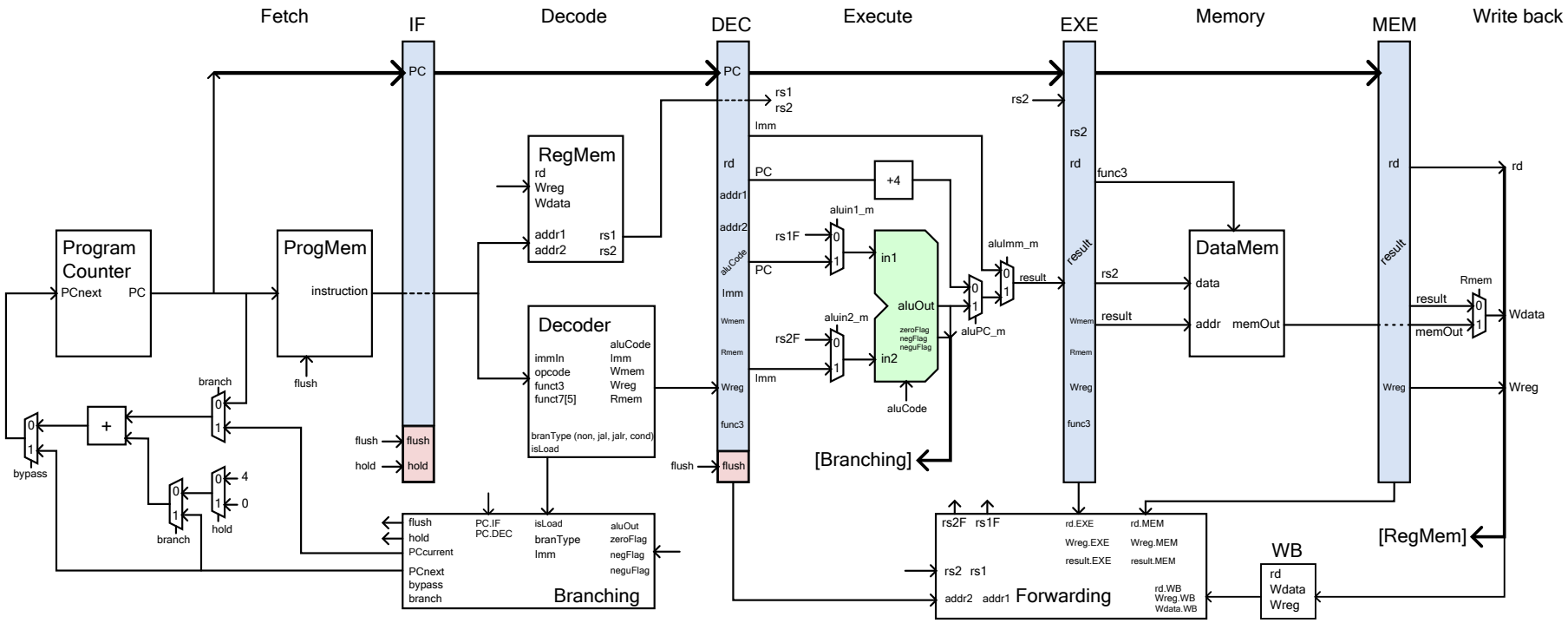

Figure 2: Pipelined RISC-V datapath

1.2 Branching and branch prediction

Another key area of complexity in pipelined processors is how branching is handled.

The outcome of conditional branch instructions (BGE, BLT, etc.) will not be knows until the execute stage, leaving cycles where instructions can be loaded uncertain.

The most simple solution is to stall (Load null instructions) until the execution phase, however this leaves empty cycles. We can do better.

Branch prediction uses an algorithm to determine whether we use these cycles predicting a failure or a success from the conditional branch. Instructions can then be processed until known in the execute phase.

If they are successes, then there is nothing to do. If they fail, then the loaded instructions need to be flushed and the correct instructions loaded in.

2 Exercises

This labs aims to take you through the implementation of forwarding and branching in in RISC-V cores. Partial complete code can be found inside of `lab3/work`. Completion of a fully functional processor will be the output for this lab.

Forwarding occurs in the Execute stage, and requires 6 overall forwarding paths for the 3 possible values to rs1F and rs2F.

2.1 Forwarding module

There is a module `lab3/work/forwarding.sv` which takes inputs :

Forwarding header

```
module forwarding(
  output logic [31:0] rs1F, rs2F,
  input [31:0] rs1, rs2,
  input [4:0] rd_EXE, rd_MEM, rd_WB,
  input [31:0] data_EXE, data_MEM, data_WB,
  input Wreg_EXE, Wreg_MEM, Wreg_WB
);
timeunit 1ns; timeprecision 100ps;
// Your code here
endmodule
```

2.2 Simple branch prediction

The simplest form of prediction is to always predict a fail, and load instructions assuming the condition will be false, achieving a 50% success rate.

Other methods exist, though this exercise will be graded based on the success rate of the implementation.

Branching header

```
module forwarding(
output logic PC,
output logic branch,
output logic hold,
output logic [31:0] PCnext,
input [2:0] func3,
input [31:0] PC_IF,
input [31:0] PC_DEC,
input is_load,
input [2:0] branch_type,
input [31:0] imm,
input [31:0] aluOut,
input zeroFlag,
input negFlag,
input neguFlag
);
  timeunit 1ns; timeprecision 100ps;
  // Your code here
endmodule
```

branch_type	Description
00	NONE
01	JAL
10	JALR
11	CONDITIONAL

func3	Conditional type
000	BGE
001	BNE
100	BLT
101	BGE
110	BLTU
111	BGEU

Specification:

- NONE instructions should update PC = PC+4 as well as set the control signals. In the event of a Load, A stall of 1 is required to ensure the next instruction can access data loaded from the memory.
- JAL instructions are guaranteed to branch. The relative offset is given in the instruction, in the form of immediate.
- JALR is guaranteed to branch. The destination is calculated in the execute stage, and can be accessed through alu_out.
- CONDITIONAL instructions are not guaranteed. It is the main challenge of this exercise to handle this case.

Conditional branching:
Level 1: Stall until the execute phase, making a functional processor.
Level 2: Implement a predict Fail branch predictor that will flush the pipeline should it be incorrect.
Level 4: Implement a flip on miss branch predictor, that will change its prediction upon failure.
Level 5: Research your own algorithm for branch prediction and implement it.
Marks will be awarded for the least cycles taken for numerous C test programs.

Important notes

- Flushing or stalling will produce addi x0 x0 0 instructions, passing through the CPU with no effect.