Lab Assignment #5

EE-25/CS-46: Computer Organization
Professor: Mark Hempstead **TAs:** Furkan Sarikaya, Jacob Carulli, and Kinan Rabbat
Tufts University, Fall 2024

Due as per the class calendar, via 'provide'

The ultimate goal of this project is to design a pipelined version of an ARM processor that can detect control and data hazards. The ARM processor is composed of components such as the Program Counter, Register File, ALU, and Data Memory, among others. Across the course of the semester, you will design each of these components individually, and then put them together to form the complete processor. Thus, it is important to understand how each component works individually as well as where it fits into the processor as a whole.

The functionality of the processor and its components should match the descriptions in the textbook unless otherwise noted. All work must be your own; copying of code will result in a zero for the project and a report to the administration.

Overview

List of assignments

- Lab 0: Set up GHDL, simulate an AND gate with 2 inputs and 1 output
- Lab 1: Basic Processor Components and Testbenches
- Lab 2: Remaining Processor Components including ALU, Memories, and control logic
- Lab 3: Implementation LEGv8: a single cycle processor that executes a subset of ARM v8-64bit ISA
- Lab 4: Pipelined processor with no hardware hazard detection
- **Lab 5: Overcoming data-hazards using forwarding and stalling**

Lab Submission

Please submit your VHDL files, Makefile, *and* a PDF report via 'provide' command on the EE/CS machines. Please follow the instructions below to use Provide and pay attention to messages you get when you try to provide.
VHDL Files: Submit the VHDL source files and any dependencies thereof. Use the starter files provided in assignment5.zip. Do not change the filenames, entity names, or input/output names. If you do, your file will not run with our tests.[1]
Makefile: If a Makefile is provided in assignment5.zip, please use it. The Makefile contains targets for each entity (inspect the Makefile to see how it works). Run `make target_name` where target name is one of the targets in the Makefile. Do not modify this Makefile except where instructed to do so. If you aren't sure, please ask. *Include this Makefile with your submission.* If you don't the course staff does not how to run your code and cannot grade you!
Report: Submit your report as a PDF**(*.pdf)** via Canvas. Demonstrate the functionality of your code by providing waveforms as detailed in the Deliverables Section. Label/annotate important signals and events in your waveforms and then provide a brief description of what is happening.

Lab 5 Starter Files

- `pipelinedcpu1.vhd`: the top-level VHDL file where you will implement the pipelined CPU.
- `pipecpu1_tb.vhd`: the VHDL file where you should write your testbench for the pipelined CPU.
- `imem_p1.vhd` and `imem_p2.vhd`: use these instruction memories to test your CPU with the two programs.

[1]Submissions that fail to to follow any of these directions may be penalized at the discretion of the grader. If you have questions, contact the TA.

- `dmem.vhd`: use this data memory when testing. It is initialized with the correct values in order for the test program to work.
- `registers.vhd`: use this register file when testing. It is also initialized with the correct values.

Lab 5 Objectives

- For this lab we do not provide a starter Makefile; you will be responsible for completing your own. You can keep your existing Makefile from Lab4. Just make sure it contains the following targets:
  - `p1`, which uses `imem_p1.vhd` to execute the test program `p1`.
  - `p2`, which uses `imem_p2.vhd` to execute the test program `p2`.
- Implement forwarding circuitry as shown in Figures 1 and 2. As we discussed in class, this circuit detects a data-hazard and forwards data from the EX/MEM and MEM/WB registers to the inputs of the ALU.
- Implement Hazard Detection Unit as shown in Figure 1. As we discussed in class, in the case of a load-use data hazard the front of the pipelined is stalled and a *nop* is inserted between the two instructions.
- Demonstrate functionality by running the test programs with data-hazards as defined later in this document.

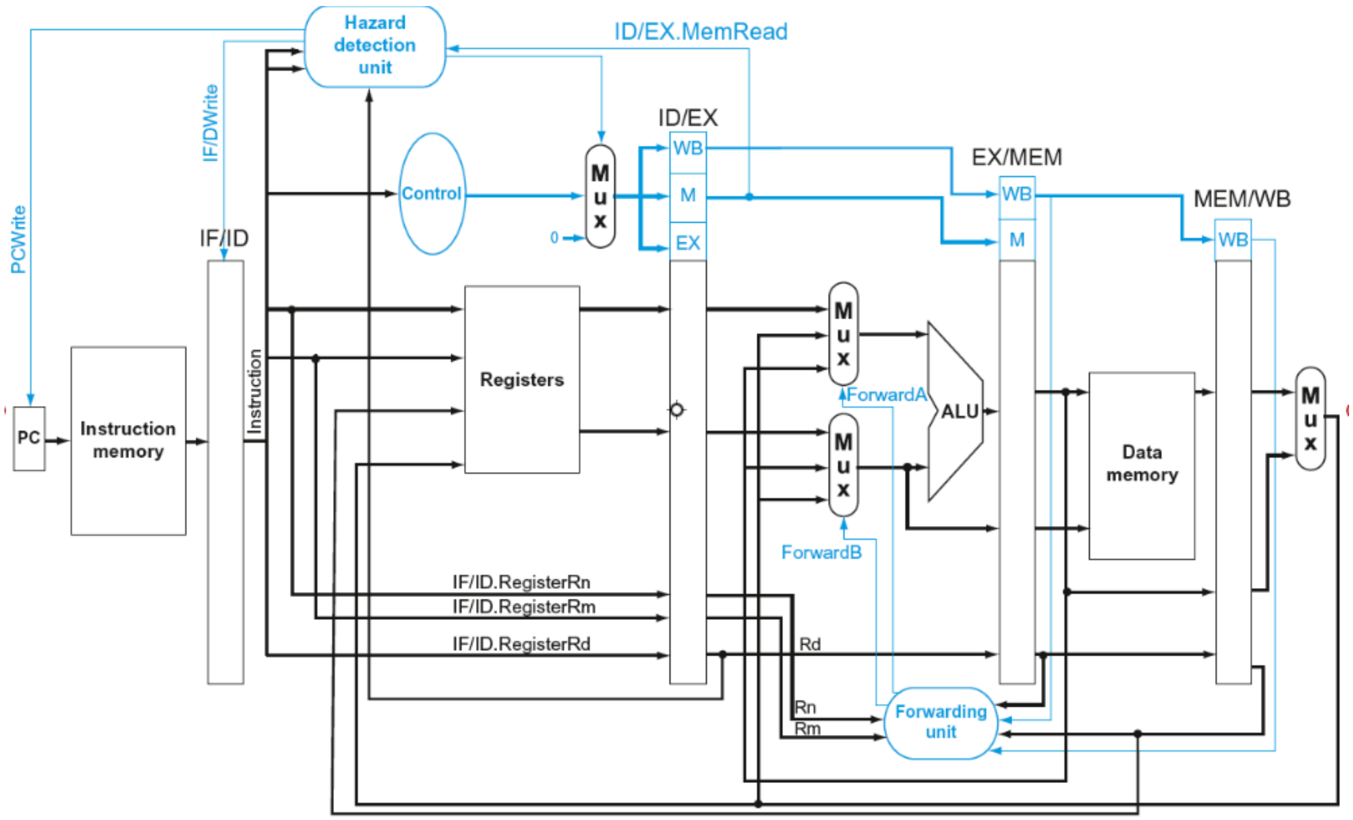


Figure 1: **Pipelined control overview showing the forwarding and hazard detection logic.** Note that this diagram leaves out some of the details of the full datapath. [Figure 4.7.4 from the textbook]

Deliverables

VHDL Files

PipelinedCPU1 and all of the entities it uses.

Report

Provide waveforms—along with annotations, labels, and descriptions—that show the successful execution of the programs defined later in this document. Be sure to:

- Include the most relevant signals in your waveform, including those related to overcoming data hazards

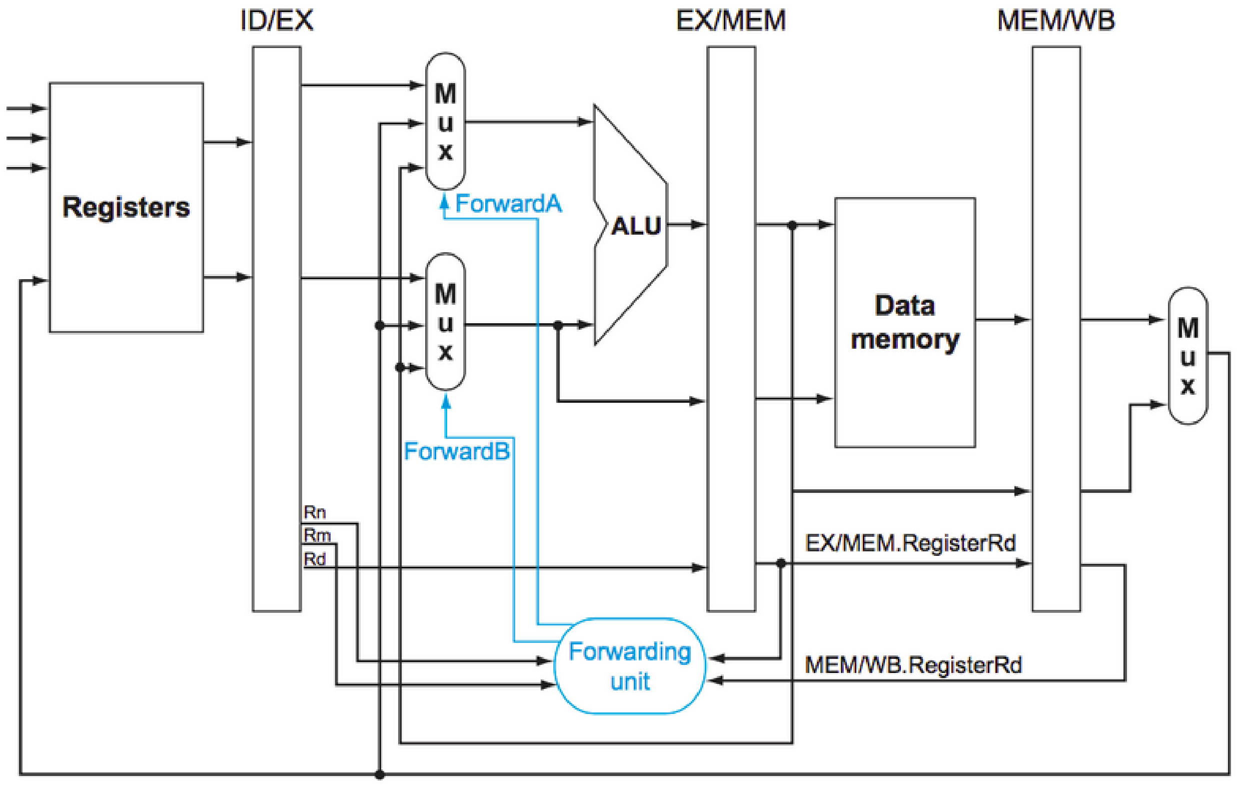


Figure 2: **Close-up of the forwarding circuitry in the datapath of Figure 1.** [Figure 4.7.2.B from the textbook]

- Point out key events such as
  - When the pipeline is stalled
  - When values are forwarded
- Clearly show what pipeline-stage each instruction is in
- Test program P2 is designed to test the CBNZ instruction and point out some problems with your implementation. Explain in your report the addition hazard present that is not handled yet by this implementation.

Extra Credit

Consider running the test program with a *nop* inserted after the first instruction ($LDUR$). Describe how the execution of that program would differ from the test program and explain *why* those differences would occur. Is there a better way to improve performance? In your explanation, include any differences, including those related to:

- stalling the pipeline
- forwarding values
- number of cycles needed to finish the program
- value of PC when the final $STUR$ occurs
- the state of DMEM/Registers at the end of the program
- Use all 0s for NOPs

Report Format:

1. Your name and the assignment name
2. A brief introduction section describing the objective of the lab and what you were building
3. A brief section describing your design, how you build your logic
4. A brief section describing your experimental setup (e.g. test benches), how you tested your logic/system

5. Results, and waveforms showing that your design works (or didn't work), and what you tried to figure out the problem if it is not working.
6. Referral and comment on waveforms and results
7. Figure & table captions (e.g. Figure #: Figure title)

ForwardingUnit Correction

The pseudo-code that describes part of the functionality of the ForwardingUnit on its section 4.7 page 322 in the ARM textbook. The corrected pseudo-code is shown below with the corrections highlighed:

```
if (MEM/WB.Regwrite
    and ( MEM/WB.RegisterRd $\neq$ 11111)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq$ 11111)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.Regwrite
    and (MEM/WB.RegisterRd $\neq$ 11111)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq$ 11111)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Program and Simulation Specifications

Test Program P1 (IMEM contents): **partial machine code | raw assembly code**

```
LDUR  X9, [XZR, 0]          11111000010000000000000111101001
ADD   X9, X9, X9            10001011000010010000000100101001
ADD   X10, X9, X9           10001011000010010000000100101010
SUB   X11, X10, X9          11001011000010010000000101001011
STUR  X11, [XZR, 8]         11111000000000001000001111101011
STUR  X11, [XZR, 16]        11111000000000010000000111101011
NOP                         00000000000000000000000000000000
NOP                         00000000000000000000000000000000
NOP                         00000000000000000000000000000000
NOP                         00000000000000000000000000000000
```

Test Program P2 (IMEM contents): **partial machine code | raw assembly code**

```
CBNZ X9 2            10110101000000000000000001001001
LSR X10, X10, 2      11010011010000000000100101001010
ANDI X11, X10, 15    10010010000000000011110101001011
LSL X12, X12, 2      11010011011000000000100110001100
ORR X21, X19, X20    10101010000101000000001001110101
SUBI X22, X21, 15    11010001000000000011111010110110
```

Initial Registers

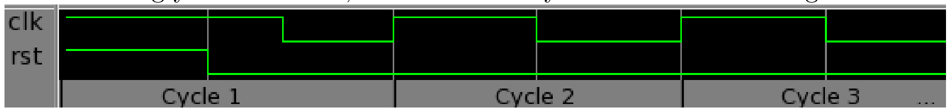
```
$x9 =  1    $x19 = 0
$x10 = 0    $x20 = 0
$x11 = 0    $x21 = 0
$x12 = 0    $x22 = 0
```

Initial DMEM

```
DMEM(0x0)  = 0x00000000
00 00 00 01
DMEM(0x8)  = 0x00000000
00 00 00 00
DMEM(0x10)  = 0x00000000
00 00 00 00
DMEM(0x18) = 0x00000000
00 00 00 00
DMEM(0x2E) = 0x00000000
00 00 00 01
```

Reset Sequence

When writing your testbench, make sure that your reset and clock signals are initialized like this:



The number of clock cycles you will need to simulate will depend on how many instructions there are in IMEM.

Partial Expected results

```
----------------INSTRUCTION[7]----------------
PC: 0000000000000014 [write_enable:1]

INSTRUCTION: F80103EB

FORWARDA              : 2

FORWARDB              : 1

------MEMORY-UPDATES-DURING-CYCLE[7]------

$t0 -> 0000000000000002
```

Entity Descriptions **(provided in assignment5.zip)**

Note that there are DEBUG ports again. There are **additional** ports compared to those of PipelineCPU0!
Pipelined CPU 1

```
entity PipelinedCPU1 is
port(
clk :in std_logic;
rst :in std_logic;
--Probe ports used for testing
-- Forwarding control signals
DEBUG_FORWARDA : out std_logic_vector(1 downto 0);
DEBUG_FORWARDB : out std_logic_vector(1 downto 0);
--The current address (AddressOut from the PC)
DEBUG_PC : out std_logic_vector(63 downto 0);
--Value of PC.write_enable
DEBUG_PC_WRITE_ENABLE : out STD_LOGIC;
--The current instruction (Instruction output of IMEM)
DEBUG_INSTRUCTION : out std_logic_vector(31 downto 0);
--DEBUG ports from other components
DEBUG_TMP_REGS : out std_logic_vector(64*4-1 downto 0);
DEBUG_SAVED_REGS : out std_logic_vector(64*4-1 downto 0);
DEBUG_MEM_CONTENTS : out std_logic_vector(64*4-1 downto 0)
);
end PipelinedCPU1;
```