EXPERIMENT 4 PRELIMINARY WORK

Yusuf Baran 2574747

1.1 Done

1.2.1

1. I implemented the datapath successfully. However, this time there are more changes comparing with the previous implementations. So that I am unable to provide a block diagram.

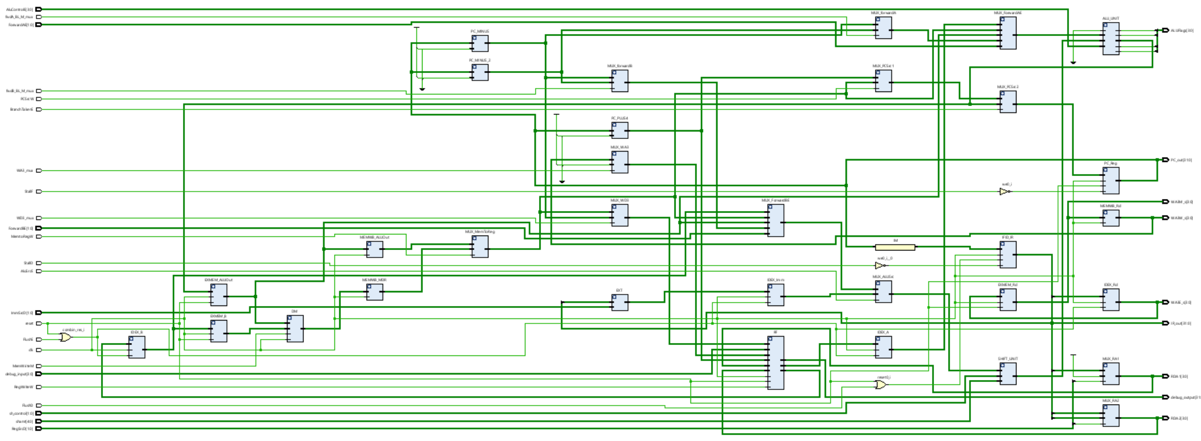


Figure 1. RTL view of the datapath

2.

Register-Shifted and Immediate-Shifted Operands

ARM allows the second ALU operand (Rm or an immediate) to be shifted before the operation.

1. **Shift Control & Amount**
    - The 2-bit sh_control and 5-bit shamt come straight from the **ID stage** (IR fields) and are captured in the ID/EX pipeline registers.
2. **Selecting Between Rm and Immediate**
    - In **EX stage**, the AluSrcE signal
    - EXB0 itself comes through the 4:1 forwarding MUX (MUX_ForwardBE), so even a shifted value can be forwarded from a later stage if needed.
3. **Shifter Module**

- SH_out feeds into the ALU's B port.

4. **ALU Integration**
   - The ALU control (AluControlE) is set by controller to one of the data-processing ops (ADD, AND, etc.) or to a "pass B" mode when you just want the shifted value.

---

2. MOV (Register & Immediate)

ARM's MOV is really just a data-processing instruction that writes a value into Rd without arithmetic:

- **Register MOV** (MOV Rd, Rm[, shift]):
  - Controller sets AluControlE = PASS_B.
  - RegSrcD chooses Rm's register-file port.
  - The shifter handles any optional LSL, LSR, etc.
  - The result is ALUResultE = SH_out and flows through to writeback.
- **Immediate MOV** (MOV Rd, #imm):
  - ImmSrcD selects the proper zero- or sign-extended immediate in the Extender.
  - AluSrcE = 1 so SH_in = ID_EX_Imm.
  - ALU still in PASS_B mode, so ALUResultE = immediate.

---

3. BL (Branch with Link)

BL must both store the return address in R14 and jump:

Link-Register Write

- For a BL, the controller asserts WA3_mux=1, WD3_mux=1, and RegWriteW=1.

2. **Branch Target**
   - Simultaneously, the ALU in EX stage computes PC + offset.
   - We complete the branching in EX for B, BL,BX whereas we do the same thing in WB for direct data assignments to PC. One reminder, even though

we do the branching in EX for BL, we wait until WB to record the PC + 4 to R14.

---

4. BX (Branch and Exchange)

For BX, the goal is simply to load the PC from whatever value is held in register Rm. In five-stage pipeline this works without adding any new hardware beyond what we already have for branches and forwarding:

1. During decode we read the register-file port for Rm just as we do for any data-processing instruction.
2. In the execute stage we put the ALU into a “pass through” mode so that whatever appears on its B input (the forwarded Rm) comes straight out as the result.
3. We assert the branch-taken signal in the following cycle, causing the PC-update multiplexers to select that ALU output as the next PC value.
4. Thanks to our existing forwarding network, if Rm was being written by an immediately preceding instruction, its new value is steered into the ALU input in time.

1.2.2

1. We implemented the controller without any problem.

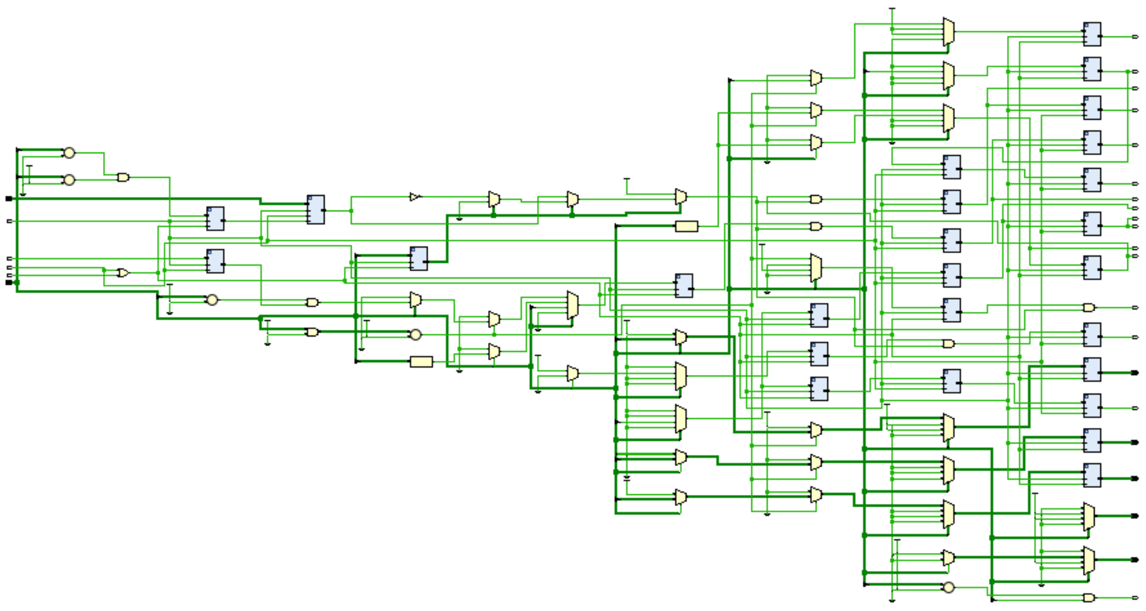


Figure 2. RTL view of the controller

2.

Register-Shifted and Immediate-Shifted Operands
ARM lets us to apply a shift to our second operand before the ALU does its work. To support that, we fed both the register's B-output and any extended immediate into a small shifter unit ahead of the ALU. The controller grabs the two-bit shift type and five-bit amount straight from the instruction in Decode and carries them forward in our ID/EX registers. In the Execute stage we choose, via AluSrcE, whether the ALU's B-input comes from that shifter or directly from the register. Because our shifter sits before the ALU and is behind our normal forwarding MUXes, we automatically get both register-shifted and immediate-shifted values forwarded correctly when there are data hazards. Also controller adjusts itself to select correct bits from instructions for immediate and register shifter operations. Specific bits are different in the instruction word for both.

MOV (Register and Immediate)
MOV is nothing more than a data-processing instruction in which the ALU is told to "pass through" its B-input. For a register MOV, we select the register file port as usual and leave our shifter control at "no shift" (or use the shifter if the MOV itself specifies a shift). For an immediate MOV, we switch in the extended immediate and still use the "pass B" ALU mode. Writeback then routes that ALU output back into Rd exactly as with any other data-processing result.

BL (Branch with Link)
BL has two effects in one instruction: it writes the return address (PC+4) into R14, and it jumps to a new target. We handled the link-register write by adding two very small MUXes at the writeback stage: one to choose R14 versus the normal Rd as the destination, and one to choose PC+4 versus the ALU or memory result as the data to write. When the controller recognizes a BL opcode it simply asserts those writeback selects and enables RegWrite. On the same cycle, it drives the ALU in Execute to compute PC+offset and raises the Branch signal so that in Fetch the PC is redirected to that target.

BX (Branch and Exchange)
BX is even simpler: we just load PC from whatever register Rm contains. In Decode we detect the BX pattern and set Branch true but disable any register write. In Execute we put the ALU into "pass-through" mode so its output is exactly Rm (after forwarding, if necessary). The next cycle, BranchTaken causes our PC-mux network to load that ALU result as the new PC. No new datapath blocks are needed—just the existing register file read, the ALU's pass mode, our forwarding MUXes, and the normal branch-target multiplexer.

1.2.3

1. We successfully handled the hazard unit.

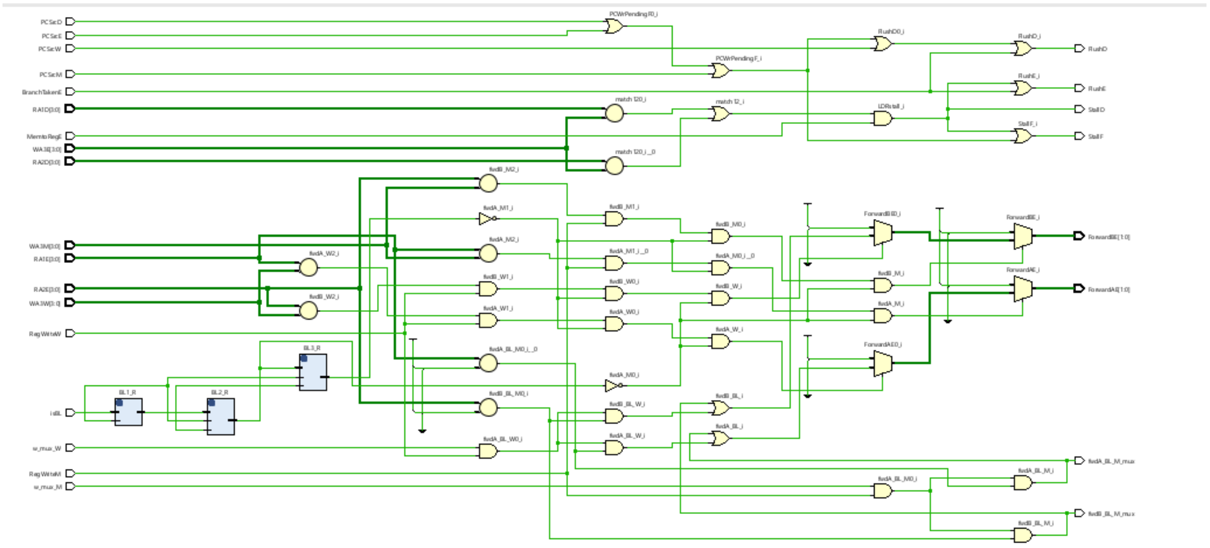


Figure 3. Hazard unit RTL view

2.

Register-shifted and immediate-shifted operands

We treat any shifted operand exactly like an ordinary register hazard. The comparison of decode-stage sources (RA1D/RA2D) against execute- and memory-stage destinations (WA3E, WA3M) is unchanged, so a value that is still in flight through the pipeline gets steered via the 4→1 forwarding MUX into the shifter and then into the ALU. No special stall or flush is needed beyond the normal load-use check.

MOV (register or immediate)

MOV is just a data-processing operation in “pass B” mode. We let it ride through the same hazard logic used by ADD, AND, etc., and forward any required operand the same way. Because it never writes to PC except in the R15-write case, no extra control-hazard logic is needed beyond the standard check for writes to R15.

BL (Branch with Link)

A BL both writes PC+4 into R14 and then jumps. To handle hazards around that link-register write we:

- Introduce three small flip-flops (BL1, BL2, BL3) so that the returned address isn’t forwarded too early or at the wrong cycle.
- Define two new forward conditions (fwdA_BL_M_mux, fwdB_BL_M_mux) that detect when the memory stage is writing PC+4 into R14. When one of those fires, we select “11” on the forwarding MUX, which injects that link value directly into the EX-stage operand.
- Stall fetch/decode for one cycle (via StallF/StallD) whenever a BL is in flight to avoid using stale data, and flush the EX stage (FlushE) on branch resolution so that no spuriously fetched instructions proceed.

- Also in controller we have a special logic for RegWrite signal because when we FlushD we generate a new instruction with RegWrite signal is 1 which isn't correct.

  BX (Branch and Exchange)

BX simply reloads PC from Rm. We handle it as a branch in decode:

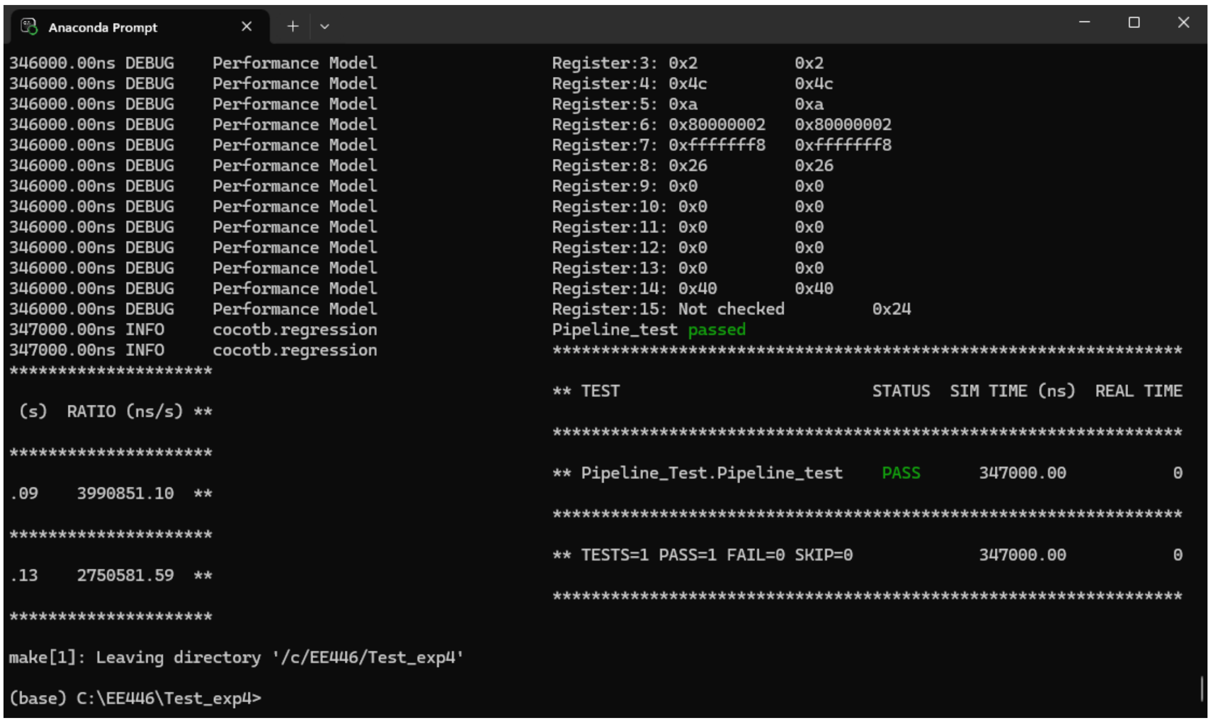
- BranchTakenE drives the standard flush signals so that any instructions fetched after the BX are thrown away.
- No register gets written by BX, so the only hazard work is to flush and then let the ALU's pass-through output (which is simply the forwarded Rm) become the new PC in the next cycle.

1.2.4

I successfully completed this step.

1.2.5

I passed the testbench.



```
Anaconda Prompt
346000.00ns DEBUG    Performance Model                  Register:3: 0x2          0x2
346000.00ns DEBUG    Performance Model                  Register:4: 0x4c         0x4c
346000.00ns DEBUG    Performance Model                  Register:5: 0xa          0xa
346000.00ns DEBUG    Performance Model                  Register:6: 0x80000002   0x80000002
346000.00ns DEBUG    Performance Model                  Register:7: 0xfffffff8   0xfffffff8
346000.00ns DEBUG    Performance Model                  Register:8: 0x26         0x26
346000.00ns DEBUG    Performance Model                  Register:9: 0x0          0x0
346000.00ns DEBUG    Performance Model                  Register:10: 0x0         0x0
346000.00ns DEBUG    Performance Model                  Register:11: 0x0         0x0
346000.00ns DEBUG    Performance Model                  Register:12: 0x0         0x0
346000.00ns DEBUG    Performance Model                  Register:13: 0x0         0x0
346000.00ns DEBUG    Performance Model                  Register:14: 0x40        0x40
346000.00ns DEBUG    Performance Model                  Register:15: Not checked         0x24
347000.00ns INFO     cocotb.regression                  Pipeline_test passed
347000.00ns INFO     cocotb.regression                  *****************************************************************
*********************
                                                        ** TEST                         STATUS  SIM TIME (ns)  REAL TIME
 (s)  RATIO (ns/s) **
                                                        *****************************************************************
*********************
                                                        ** Pipeline_Test.Pipeline_test     PASS      347000.00           0
.09    3990851.10  **
                                                        *****************************************************************
*********************
                                                        ** TESTS=1 PASS=1 FAIL=0 SKIP=0           347000.00           0
.13    2750581.59  **
                                                        *****************************************************************
*********************

make[1]: Leaving directory '/c/EE446/Test_exp4'

(base) C:\EE446\Test_exp4>
```

Figure 4. Cocotb result