

End-semester Assignment

Neelam Sharma (18307R030)
EE 789: Algorithmic Design of Digital Systems

August 15, 2020

1 Introduction

A 32-bit 7-staged pipelined processor is designed in Aa language to execute a custom instruction set.

1.1 Instruction set

1. HALT
2. SBIR imm rd
Loads byte imm into lowest byte of rd.
3. LOAD rs1 rd
 $\langle rd \rangle \leftarrow mem[\langle rs1 \rangle]$
4. STORE rs1 rs2
 $mem[\langle rs1 \rangle] \leftarrow \langle rs2 \rangle$
5. OP rs1 rs2 rd
 $\langle rd \rangle \leftarrow \langle rs1 \rangle \text{ OP } \langle rs2 \rangle$, where OP = AND, OR, XNOR, XOR, ADD, SUB, SLL, SRL, SRA
6. BZ rs1 rs2
if $\langle rs1 \rangle == 0$ jump to $\langle rs2 \rangle$
7. BN rs1 rs2
if $\langle rs1 \rangle$ is negative, jump to $\langle rs2 \rangle$
8. CALL rs1 rd
jump to $\langle rs1 \rangle$, store pc in rd.
9. JMP rs1 *jump to $\langle rs1 \rangle$*
10. CMP rs1 rs2 rd
 $\langle rd \rangle \leftarrow (excmux$
 $(\langle rs1 \rangle == \langle rs2 \rangle) 0$
 $(\langle rs1 \rangle < \langle rs2 \rangle) -1$
 $(\langle rs1 \rangle > \langle rs2 \rangle) 1)$

2 Architectural overview

This processor includes following stages:

1. Fetch stage
2. Instruction cache stage
3. Decode
4. Register read
5. Execute
6. Memory read/write
7. Write-back

3 Design Decisions

3.1 Forwarding Decisions

1. Data cache state to execute stage:
Condition 1: $dcache.reg_write == 1$ $dcache.rd == iexec.rs1$
Action: Forward $dcache.exec_result$ to $rs2$ input.
Condition 2: $dcache.reg_write == 1$ $dcache.rd == iexec.rs2$
Action: Forward $dcache.exec_result$ to $rs2$ input.
Example:
ADD $r1, r2, r4$
SUB $r4, r7, r9$
2. Instruction retire stage to execute stage:
Condition 1: $iretire.reg_write == 1$ $iretire.rd == iexec.rs1$
Action: Forward $iretire.mem_regData$ to $rs1$ input.
Condition 2: $iretire.reg_write == 1$ $iretire.rd == iexec.rs2$
Action: Forward $iretire.mem_regData$ to $rs2$ input.
Example:
OR $r5, r2, r4$
AND $r5, r6, r7$
SUB $r4, r7, r9$
3. Instruction retire stage to Data cache stage:
Condition 1: $iretire.reg_write == 1$ $iretire.rd == dcache.memAddr$
Action: Forward $iretire.mem_regData$ to $dcache.memAddr$
Condition 2: $iretire.reg_write == 1$ $iretire.rd == dcache.rs1$
Action: Forward $iretire.mem_regData$ to $dcache.dataIn$
Example:
LOAD $r2, r0, r5$
STORE $r5, r2, r8$

4. Instruction retire stage to Register file stage:
 Condition 1: *iretire.reg_write* == 1 *iretire.rd* == *regFile.rs1*
 Action: Forward *iretire.mem_regData* to rd1 output of register file stage.
 Condition 2: *iretire.reg_write* == 1 *iretire.rd* == *regFile.rs2*
 Action: Forward *iretire.mem_regData* to rd2 output of register file stage.
 Example:
 OR r5, r2, r4
 AND r5, r6, r7
 SUB r8, r7, r9
 SLL r4, r11, r9

3.2 Branch hazard Decisions

1. BN, BZ instructions are detected at the output of EX/DCACHE pipelined register. Five stages (Instruction Fetch, Instruction cache, Decode, Register file, Execute) are flushed for a conditional branch instruction.
 Condition: *dcache.is_Branch* == 1
 Action: Flush the stages previous to Data cache stage.
2. Unconditional jumps are detected at the output of REG/EX pipelined register. Four stages (Instruction Fetch, Instruction cache, Decode, Register file) are flushed for a unconditional branch instruction.
 Condition: *iexec.is_unconditional_Branch* == 1
 Action: Flush the stages previous to Execute stage.

3.3 Stalling Decisions

1. LOAD preceded by some dependent instruction.
 Condition: *iexec.rd* == *regFile.rs1* | *iexec.rd* == *regFile.rs2*
 Action: Stall first four stages (Instruction Fetch, Instruction cache, Decode, Register file) and flush REG/EX pipelined register in next iteration.
 Example:
 LOAD r1, r0, r4
 ADD r4, r6, r9

4 Diagram

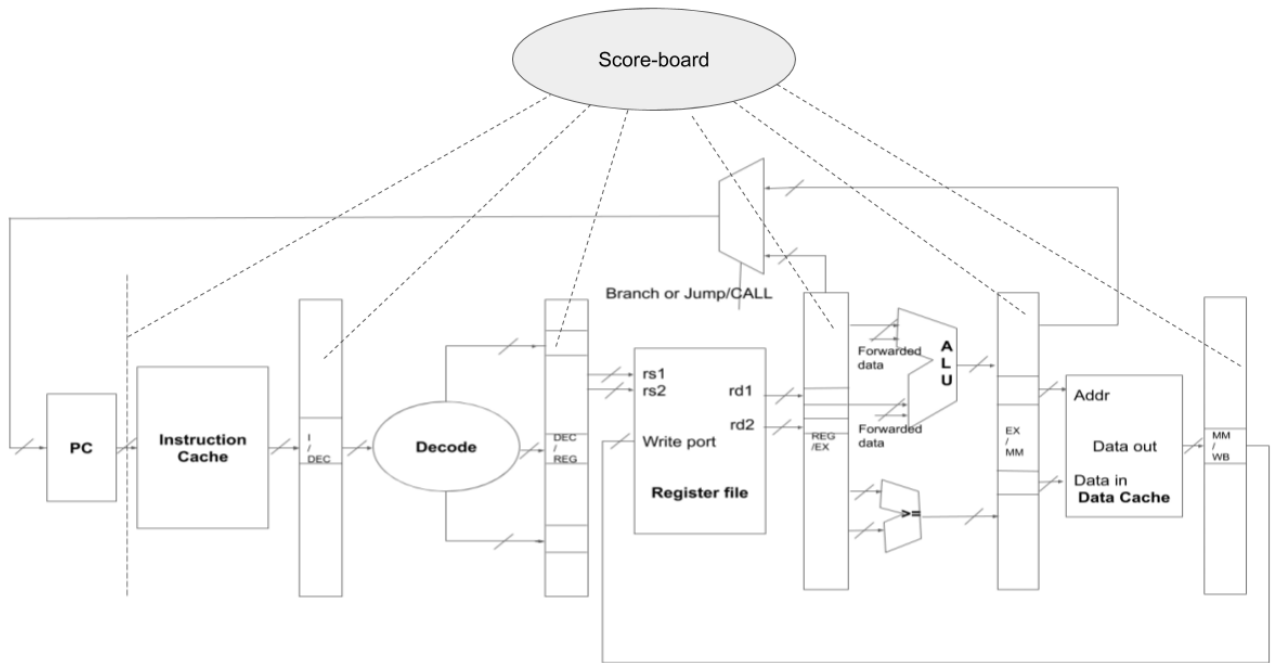


Figure 1: Pipelined architecture

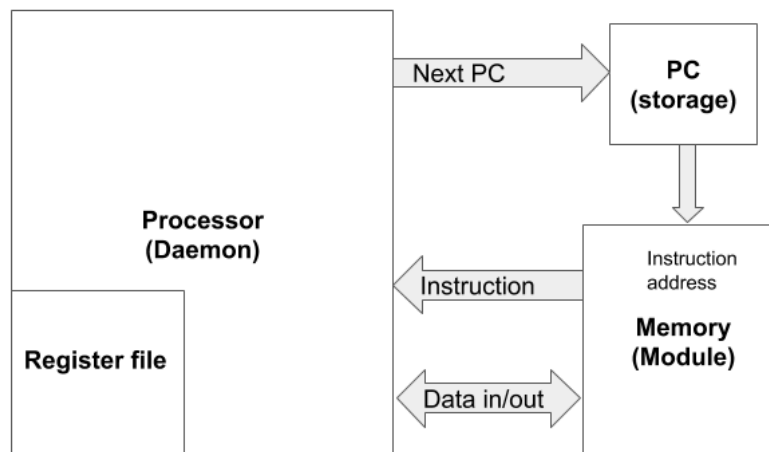


Figure 2: Overall architecture

Consider the data stored in memory with carry = 0x00000001 set initially:

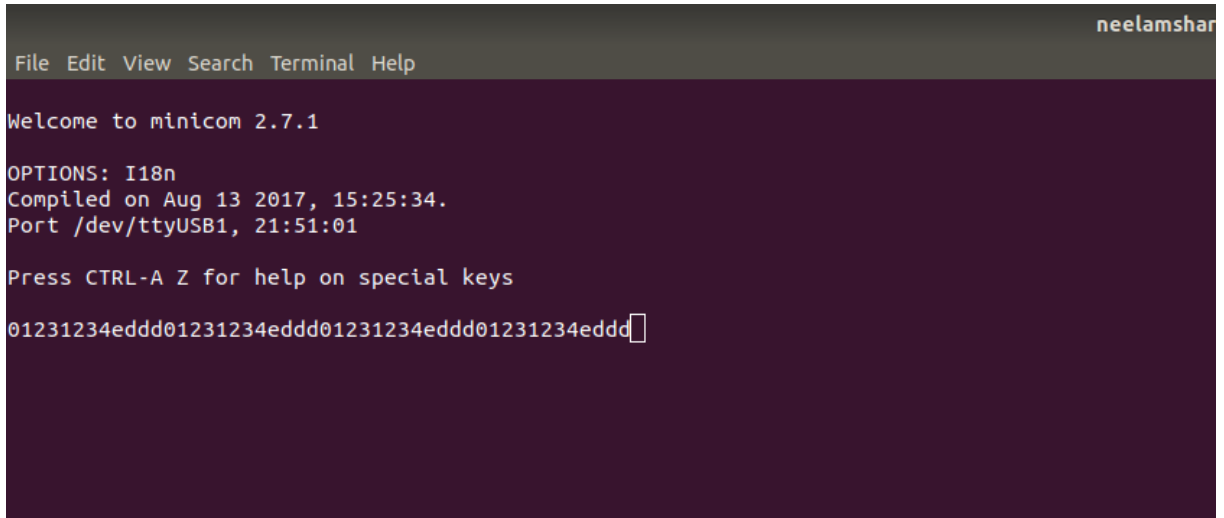
Location	Data stored	Result
0	SBIR 30 00 03	[r3] = x00000030 (ASCII value= 0)
1	SBIR 31 00 02	[r2] = x00000031 (ASCII value= 1)
2	SBIR 32 00 01	[r1] = x00000032 (ASCII value= 2)
3	SBIR 33 00 00	[r0] = 0x00000033 (ASCII value= 3)
4	SBIR 31 00 04	[r4] = 0x00000031 (ASCII value= 1)
5	SBIR 32 00 05	[r5] = 0x00000032 (ASCII value= 2)
6	SBIR 33 00 06	[r6] = 0x00000033 (ASCII value= 3)
7	SBIR 34 00 07	[r7] = 0x00000034 (ASCII value= 4)
8	ADD r0, r4, r8	[r8] = 0x00000065 (ASCII value= e)
9	ADD r1, r5, r9	[r9] = 0x00000064 (ASCII value = d)
10	ADD r2, r6, r10	[r10] = 0x00000064 (ASCII value = d)
11	ADD r3, r7, r11	[r11] = = 0x00000064 (ASCII value = d)
12	HALT	None

```
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 32
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 33
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 31
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 32
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 33
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 34
Received from UART: 0
Received from UART: 0
Received from UART: 65
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 64
Received from UART: 0
Received from UART: 0
Received from UART: 0
Received from UART: 64
Received from UART: 0
Received from UART: 0
Received from UART: 64
[1] 0:[mux]#
```

5

5.2 Serial Port results

The button (btnU) on Basys3 is used to start the processor, pressing it resulted in multiple times processor to be triggered that's why same result is repeated 4 times.



```
neelamshar
File Edit View Search Terminal Help
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB1, 21:51:01

Press CTRL-A Z for help on special keys

01231234eddd01231234eddd01231234eddd01231234eddd
```

Figure 4: Simulation Output

Explanation

- SBIR 30 00 03 sends output of data to be written on register r3 i.e., x00000030 (ASCII value=0).
- SBIR 31 00 02 sends output of data to be written on register r2 i.e., x00000031 (ASCII value=1).
- SBIR 32 00 01 sends output of data to be written on register r1 i.e., x00000032 (ASCII value=2).
- SBIR 33 00 00 sends output of data to be written on register r0 i.e., x00000033 (ASCII value=3).
- SBIR 31 00 04 sends output of data to be written on register r4 i.e., x00000031 (ASCII value=1).
- SBIR 32 00 05 sends output of data to be written on register r5 i.e., x00000032 (ASCII value=2).
- SBIR 33 00 06 sends output of data to be written on register r6 i.e., x00000033 (ASCII value=3).
- SBIR 34 00 07 sends output of data to be written on register r7 i.e., x00000034 (ASCII value=4).
- ADD r0, r4, r8 sends output of data to be written on register r8 i.e., x00000065 (ASCII value=e).

- ADD r1, r5, r9 sends output of data to be written on register r9 i.e., x00000064 (ASCII value=d).
- ADD r2, r6, r10 sends output of data to be written on register r10 i.e., x00000064 (ASCII value=d).
- ADD r3, r7, r11 sends output of data to be written on register r11 i.e., x00000064 (ASCII value=d).