EE309 Project
Pipelined RISC Processor

Mihir Kavishwar (17D070004)
Rishabh Dahale (17D070008)
Mithilesh Vaidya (17D070011)
Anubhav Agarwal (17D070026)

1 Datapath

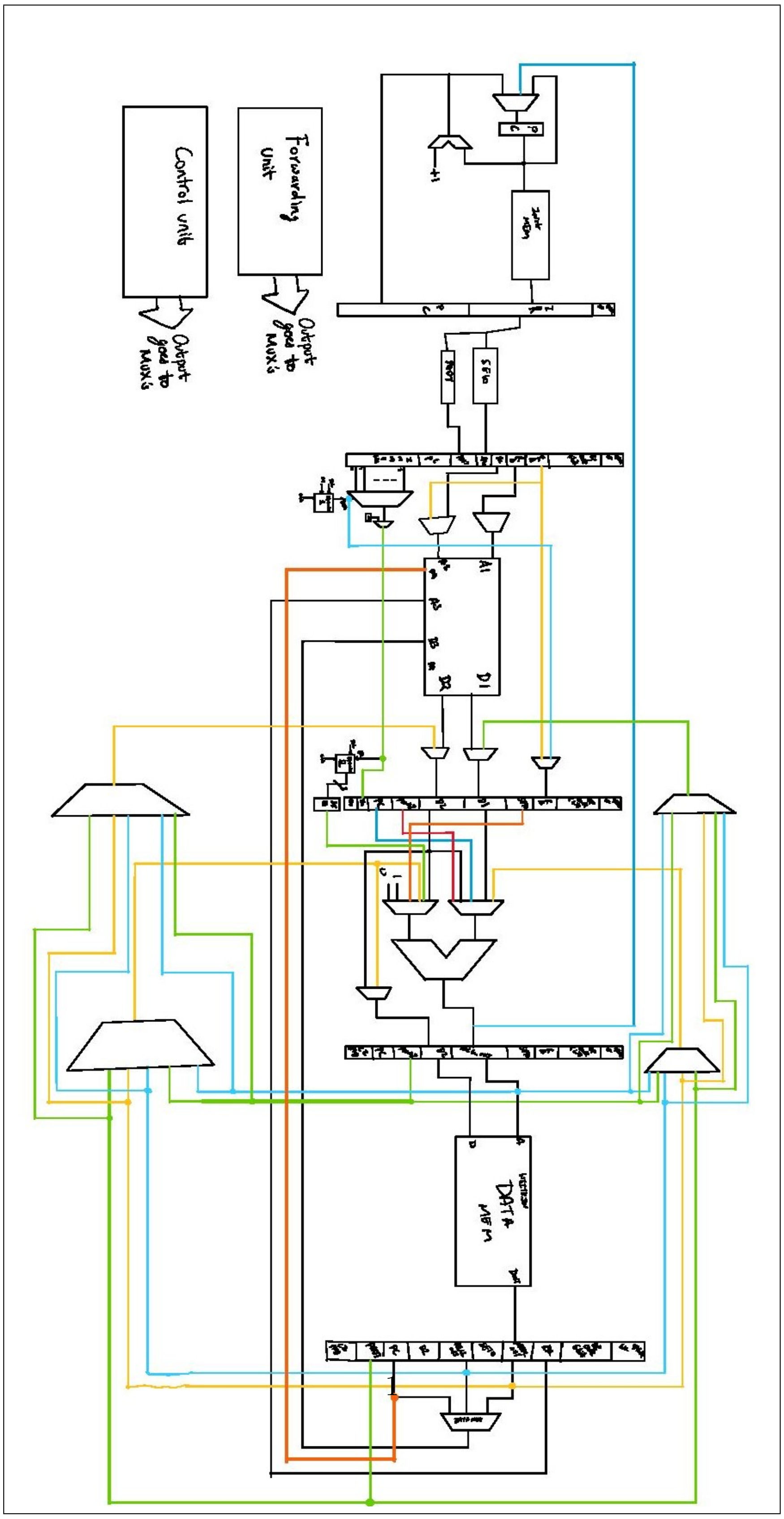

Figure 1: Complete Datapath

2 Design Description

2.1 IF Stage

This stage contains the following:

1. Program Counter (PC)
2. Instruction Memory
3. ALU to increment PC by one unit

PC contains the address of next instruction which is fetched from the Instruction Memory and stored in Instruction Register. A MUX is used to select whether the updated PC is obtained by incrementing the previous PC value or the from the execution stage. The IF/ID pipeline register stores the PC, Instruction Register and Update bit that indicates whether IF/ID pipeline register should be overwritten or not.

2.2 ID stage

In this stage we read the instruction from IF/ID register, split it into its different fields and store them in ID/RR register. The SE10 unit is used to extend 6 bit immediate data in I type instruction to 16 bits. The Pad7 unit is used to pad 7 bits to the left of 9 bit immediate data in LHI instruction to 16 bits. Therefore the ID/RR register contains the Opcode, RA, RB, RC, SE10 output, Pad7 output, PC and Update bit.

2.3 RR Stage

This stage contains the following:

1. Register File (RF)
2. Up-counters (UC1 and UC2)
3. Equivalence Checker

We read from register file in this stage. Input to RF_A1 and RF_A2 comes from RA, RB or RC depending on the instruction. The data that is read from RF is stored in RR/EX register.

There is also an equivalence checker which compares data at RF_D1 and RF_D2 in order the decide whether the branch should be taken in BEQ instruction. In cases on dependent instructions, the input to equivalence checker and pipeline registers comes from the forwarding path.

To implement Load Multiple and Store Multiple instructions, we are using two 3-bit up-counters. Up-counter1 keeps track of number of cycles and its overflow indicates that the process is complete. The output of the up-counter1 is used as select line for the 8-to-1 mux, where the 8 inputs come from immediate field. This helps us is serializing the parallel data. This serial data is given as enable to Up-counter2. The output of up-counter2 is added with data in RA in execution stage to calculate memory address of location from where we have to read or write. The RR/EX register contains various things as shown in the datapath which are useful for the next stages.

2.4 EX stage

This stage contains the main ALU which performs addition or NAND operation. The input to ALU can either come from RR/EX register (D1, D2, SE10, Pad7 or PC) or can come from one of the forwarding paths. The output (including C and Z flag) is stored in next pipeline register - EX/MEM. The EX/MEM register contains various things as shown in the datapath which are useful for the next stages.

2.5 MEM stage

This stage contains the Data Memory. Data is either read from this memory or written to this memory depending on the instruction. The memory address always comes from EX/MEM.ALU_out register. The data which has to be written comes from EX/MEM.D2 register. The data which is read is stored in MEM/WB.Mem_out register. The MEM/WB register contains various things as shown in the datapath which are useful for the next stage.

2.6 WB stage

After going through various stages the final data is written back to RF in this stage. The destination register in all the cases is RA. The data to be written back can come from three locations: Mem_{out}, ALU_{OUT} or PC and hence a mux has been used to select the corresponding source.

In order to store PC to register R7, we are using additional ports RF_A4 and RF_D4 in register file. The port RF_A4 is permanently mapped to 111.

The wr_{en} pin on register file controls whether data needs to be written to register file or not.

3 Forwarding Logic

The following is the pseudo code used for forwarding logic.

```
EX FORWARDING UNIT
-------------------

Current Instruction = I
Previous Instruction = Ip
Previous of previous Instruction = Ipp

I>Ip means I depends on Ip
I>Ipp means I depends on Ipp

Possible forwarding paths:

1. a) EX/MEM.ALU_out -> ALU_in1 // When (I>Ip) and (Ip uses ALU to
compute result)
b) EX/MEM.ALU_out -> ALU_in2
(This can be directly forwarded to the D1 in ex stage so the update
of further registers can also be done)

2. a) MEM/WB.ALU_out -> ALU_in1 // When (I>Ipp) and (Ipp uses ALU
to compute result)
b) MEM/WB.ALU_out -> ALU_in2

3. a) EX/MEM.PAD7 -> ALU_in1 // When (I>Ip) and (Ip is LHI)
b) EX/MEM.PAD7 -> ALU_in2

4. a) MEM/WB.PAD7 -> ALU_in1 // When (I>Ipp) and (Ipp is LHI)
b) MEM/WB.PAD7 -> ALU_in2

5. a) MEM/WB.MD_out -> ALU_in1 // When (I>Ip) and (Ip is LW)
b) MEM/WB.MD_out -> ALU_in2

MUX1
   1---|   |
   2---|  |
   3---|  |---ALU_in1_forwarded
   4---|  |
   5---|  |

    MUX2
   1---|  |
   2---|  |
   3---|  |---ALU_in2_forwarded
   4---|  |
   5---|  |
```

We always stall 1 cycle if we encounter LW so we need not check dependence on Ipp

For situations 1 to 4:
Ip is stored in EX/MEM
Ipp is stored in MEM/WB

For situation 5:
Ip is stored in MEM/WB

For Load Multiple assume that we sufficiently stall so that RF is already updated when next instruction is in RR

If logic evaluates more than one forwarding path to same ALU_in then choose the one coming due to I->Ip
(Example of when this happens:

```
Ipp - ADD R1, R2, R3
Ip  - ADD R1, R4, R5
I   - ADD R2, R1, R6
```

Here I>Ip and I>Ipp but we will only consider I>Ip
)

// INSTRUCTION FORMAT ADD RA, RB, RC => RA=RB+RC, RB always go into ALU_in1, RC always go into ALU_in2

```
CASES:
Part 1 written above
if EX/MEM.RA == RR/EX.RB:
if EX/MEM.OPCODE == ADD/ADC.(EX/MEM.C set)/ADZ.(EX/MEM.Z set)/ADI
NDU/NDC.(EX/MEM.C set)/NDZ.(EX/MEM.Z set):
if RR/EX.OPCODE == ADD/ADC/ADZ/ADI/NDU/NDC/NDZ/LW/SW/JLR
EX/MEM.ALU_out -> ALU_in1

if EX/MEM.RA == RR/EX.RC:
if EX/MEM.OPCODE == ADD/ADC.(EX/MEM.C set)/ADZ.(EX/MEM.Z set)/ADI
NDU/NDC.(EX/MEM.C set)/NDZ.(EX/MEM.Z set):
if RR/EX.OPCODE == ADD/ADC/ADZ/NDU/NDC/NDZ:
EX/MEM.ALU_out -> ALU_in2

if EX/MEM.RA == RR/EX.RA:
if EX/MEM.OPCODE == ADD/ADC.(EX/MEM.C set)/ADZ.(EX/MEM.Z set)/ADI
NDU/NDC.(EX/MEM.C set)/NDZ.(EX/MEM.Z set):
if RR/EX.OPCODE == SW/LM/SM
EX/MEM.ALU_out -> ALU_in2

Part 2 written above
if MEM/WB.RA == RR/EX.RB:
if MEM/WB.OPCODE == ADD/ADC.(MEM/WB.C set)/ADZ.(MEM/WB.Z set)/ADI
NDU/NDC.(MEM/WB.C set)/NDZ.(MEM/WB.Z set):
if RR/EX.OPCODE == ADD/ADC/ADZ/ADI/NDU/NDC/NDZ/LW/SW/JLR:
MEM/WB.ALU_out -> ALU_in1

if MEM/WB.RA == RR/EX.RC:
if MEM/WB.OPCODE == ADD/ADC.(MEM/WB.C set)/ADZ.(MEM/WB.Z set)/ADI
NDU/NDC.(MEM/WB.C set)/NDZ.(MEM/WB.Z set):
if RR/EX.OPCODE == ADD/ADC/ADZ/NDU/NDC/NDZ:
MEM/WB.ALU_out -> ALU_in2

if MEM/WB.RA == RR/EX.RA:
if EX/MEM.OPCODE == ADD/ADC.(EX/MEM.C set)/ADZ.(EX/MEM.Z set)/ADI
NDU/NDC.(EX/MEM.C set)/NDZ.(EX/MEM.Z set):
if RR/EX.OPCODE == SW/LM/SM
MEM/WB.ALU_out -> ALU_in2

Part 3 written above
if EX/MEM.RA == RR/EX.RB:
```

```
if EX/MEM.OPCODE == LHI:
if RR/EX.OPCODE == ADD/ADC/ADZ/ADI/NDU/NDC/NDZ/LW/SW/JLR:
EX/MEM.PAD7 -> ALU_in1

if EX/MEM.RA == RR/EX.RC:
if EX/MEM.OPCODE == LHI:
if RR/EX.OPCODE == ADD/ADC/ADZ/NDU/NDC/NDZ:
EX/MEM.PAD7 -> ALU_in2

if EX/MEM.RA == RR/EX.RA:
if EX/MEM.OPCODE == LHI:
if RR/EX.OPCODE == SW/LM/SM:
EX/MEM.PAD7 -> ALU_in2

Part 4 written above (Ipp is LHI)
if MEM/WB.RA == RR/EX.RB:
if MEM/WB.OPCODE == LHI:
if  RR/EX.OPCODE == ADD/ADC/ADZ/ADI/NDU/NDC/NDZ/LW/SW/JLR:
MEM/WB.PAD7 -> ALU_in1

if MEM/WB.RA == RR/EX.RC:
if MEM/WB.OPCODE == LHI:
if  RR/EX.OPCODE == ADD/ADC/ADZ/NDU/NDC/NDZ:
MEM/WB.PAD7 -> ALU_in2

if MEM/WB.RA == RR/EX.RA:
if MEM/WB.OPCODE == LHI:
if RR/EX.OPCODE == SW/LM/SM:
EX/MEM.PAD7 -> ALU_in2

Part 5 written Above
if MEM/WB.RA == RR/EX.RB:
if MEM/WB.OPCODE == LW:
if RR/EX.OPCODE == ADD/ADC/ADZ/ADI/NDU/NDC/NDZ/LW/SW/JLR:
MEM/WB.MD_out -> ALU_in1

if MEM/WB.RA == RR/EX.RC:
if MEM/WB.OPCODE == LW:
if  RR/EX.OPCODE == ADD/ADC/ADZ/NDU/NDC/NDZ:
MEM/WB.MD_out -> ALU_in2
```

```
if MEM/WB.RA == RR/EX.RA:
if MEM/WB.OPCODE == LW:
if RR/EX.OPCODE == SW/LM/SM:
MEM/WB.MD_out -> ALU_in2

**********************************************************************

RR FORWARDING UNIT
-------------------

Current Instruction = I
Previous Instruction = Ip
Previous of previous Instruction = Ipp

I>Ip means I depends on Ip
I>Ipp means I depends on Ipp

Possible forwarding paths:

1. a) InputOf(EX/MEM.ALU_out) -> Comp_in1 // When (I>Ip) and (Ip uses
ALU to compute result)
b) InputOf(EX/MEM.ALU_out) -> Comp_in2

2. a) InputOf(MEM/WB.ALU_out) -> Comp_in1 // When (I>Ipp) and (Ipp
uses ALU to compute result)
b) InputOf(MEM/WB.ALU_out) -> Comp_in2

3. a) InputOf(EX/MEM.PAD7) -> Comp_in1 // When (I>Ip) and (Ip is
LHI)
b) InputOf(EX/MEM.PAD7) -> Comp_in2

4. a) InputOf(MEM/WB.PAD7) -> Comp_in1 // When (I>Ipp) and (Ipp is
LHI)
b) InputOf(MEM/WB.PAD7) -> Comp_in2

5. a) InputOf(MEM/WB.MD_out) -> Comp_in1 // When (I>Ip) and (Ip
is LW)

MUX1
   1---|  |
   2---|  |
3---|  |---Comp_in1_forwarded
4---|  |
5---|  |

 MUX2
1---|  |
2---|  |
3---|  |---Comp_in2_forwarded
4---|  |
5---|  |
```

```
We always stall 1 cycle if we encounter LW so we need not check
dependence on Ipp

For situations 1 to 4:
Ip is stored in RR/EX
Ipp is stored in EX/MEM

For situation 5:
Ip is stored in EX/MEM

For Load Multiple assume that we sufficiently stall so that RF is
already updated when next instruction is in RR

If logic evaluates more than one forwarding path to same ALU_in then
choose the one coming due to I->Ip
(Example of when this happens:
Ipp - ADD R1, R2, R3
Ip  - ADD R1, R4, R5
I   - BEQ R3, R1, Imm
Here I>Ip and I>Ipp but we will only consider I>Ip
)


// INSTRUCTION FORMAT ADD RA, RB, RC => RA=RB+RC, RB always go into
ALU_in1, RC always go into ALU_in2

CASES:
Part 1 written above
```

```
if RR/EX.RA == ID/RR.RB:
if RR/EX.OPCODE == ADD/ADC.(InputOf(EX/MEM.C) set)/ADZ.(InputOf(EX
MEM.Z) set)/ADI/NDU/NDC.(InputOf(EX/MEM.C) set)/NDZ.(InputOf(EX
MEM.Z) set):
if ID/RR.OPCODE == BEQ:
InputOf(EX/MEM.ALU_out) -> Comp_in1

if RR/EX.RA == ID/RR.RA:
if RR/EX.OPCODE == ADD/ADC.(InputOf(EX/MEM.C) set)/ADZ.(InputOf(EX
MEM.Z) set)/ADI/NDU/NDC.(InputOf(EX/MEM.C) set)/NDZ.(InputOf(EX
MEM.Z) set):
if ID/RR.OPCODE == BEQ
InputOf(EX/MEM.ALU_out) -> Comp_in2

Part 2 written above
if EX/MEM.RA == ID/RR.RB:
if EX/MEM.OPCODE == ADD/ADC.(InputOf(MEM/WB.C) set)/ADZ
(InputOf(MEM/WB.Z) set)/ADI/NDU/NDC.(InputOf(MEM/WB.C) set)/NDZ
(InputOf(MEM/WB.Z)set):
if ID/RR.OPCODE == BEQ:
InputOf(MEM/WB.ALU_out) -> Comp_in1

if EX/MEM.RA == ID/RR.RA:
if EX/MEM.OPCODE == ADD/ADC.(InputOf(MEM/WB.C) set)/ADZ
(InputOf(MEM/WB.Z) set)/ADI/NDU/NDC.(InputOf(MEM/WB.C) set)/NDZ
(InputOf(MEM/WB.Z)set):
if ID/RR.OPCODE == BEQ:
InputOf(MEM/WB.ALU_out) -> Comp_in2

Part 3 written above
if RR/EX.RA == ID/RR.RB:
if RR/EX.OPCODE == LHI:
if ID/RR.OPCODE == BEQ:
InputOf(EX/MEM.PAD7) -> Comp_in1

if RR/EX.RA == ID/RR.RA:
if RR/EX.OPCODE == LHI:
if ID/RR.OPCODE == BEQ:
InputOf(EX/MEM.PAD7) -> Comp_in2

Part 4 written above (Ipp is LHI)
```

```
if EX/MEM.RA == ID/RR.RB:
if EX/MEM.OPCODE == LHI:
if ID/RR.OPCODE == BEQ:
InputOf(MEM/WB.PAD7) -> Comp_in1

if EX/MEM.RA == ID/RR.RA:
if EX/MEM.OPCODE == LHI:
if ID/RR.OPCODE == BEQ:
InputOf(MEM/WB.PAD7) -> Comp_in2

Part 5 written Above
if EX/MEM.RA == ID/RR.RB:
if EX/MEM.OPCODE == LW:
if ID/RR.OPCODE == BEQ:
InputOf(MEM/WB.MD_out) -> Comp_in1

if EX/MEM.RA == ID/RR.RA:
if EX/MEM.OPCODE == LW:
if ID/RR.OPCODE == BEQ:
InputOf(MEM/WB.MD_out) -> Comp_in2
```

4 Netlist

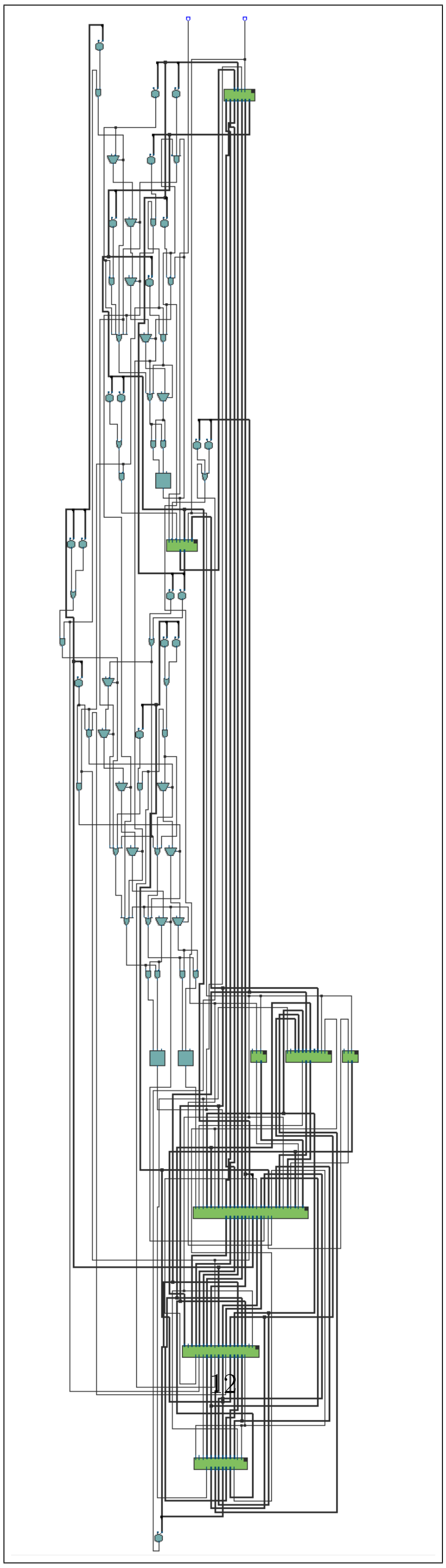
Figure 2: RTL Netlist