RISC Pipelining Project Report

Puneet Nemade, 16D070003
Harsh Deshpande, 16D070011
Viraj Nadkarni, 16D070013
Navoj Ramesh, 16D070059

December 2, 2018

1 Overview

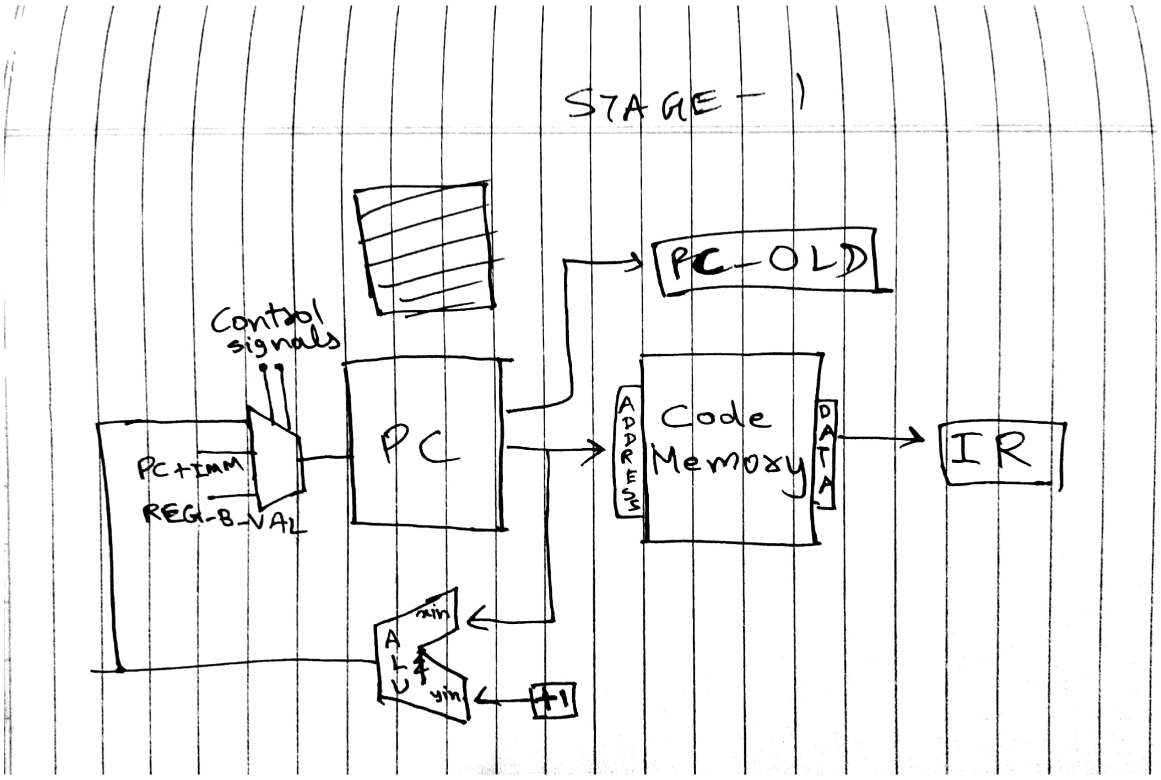
The aim of this project was to design and simulate a pipelined microprocessor capable of executing upto 19 machine-level instructions. The microprocessor designed is a 160bit microprocessor with 7 registers. The pipeline is split into 6 stages, namely :

1. Instruction fetch
2. Instruction decode
3. Register/ Operand fetch
4. Arithmetic/Logicasl computation
5. Memory read/write
6. Register write-back

All of the six stages has been explained in detail in this document. Due to being pipelined, efficient forwarding mechanisms were introduced to reduce delays and keep the CPI as close to 1 as possible. The identification and resolution of such hazards is also explained in the respective stages.

2 The Pipeline

2.1 Stage 1 - Instruction Fetch



Stage 1 of the pipeline consisted of the data path required for fetching the desired instruction from the memory. This stage decided the which instructor will be executed in the following stages. Also it received as an input that decided whether the current instruction is valid or not.
Apart from this stage 1 also contains its own ALU, that it uses to compute PC+1 and depending upon an input bit stores PC+1 or PC+Imm in the Program Counter for the next instruction. Some of the important input and output signal of the entity in the code are:

2.1.1 Inputs

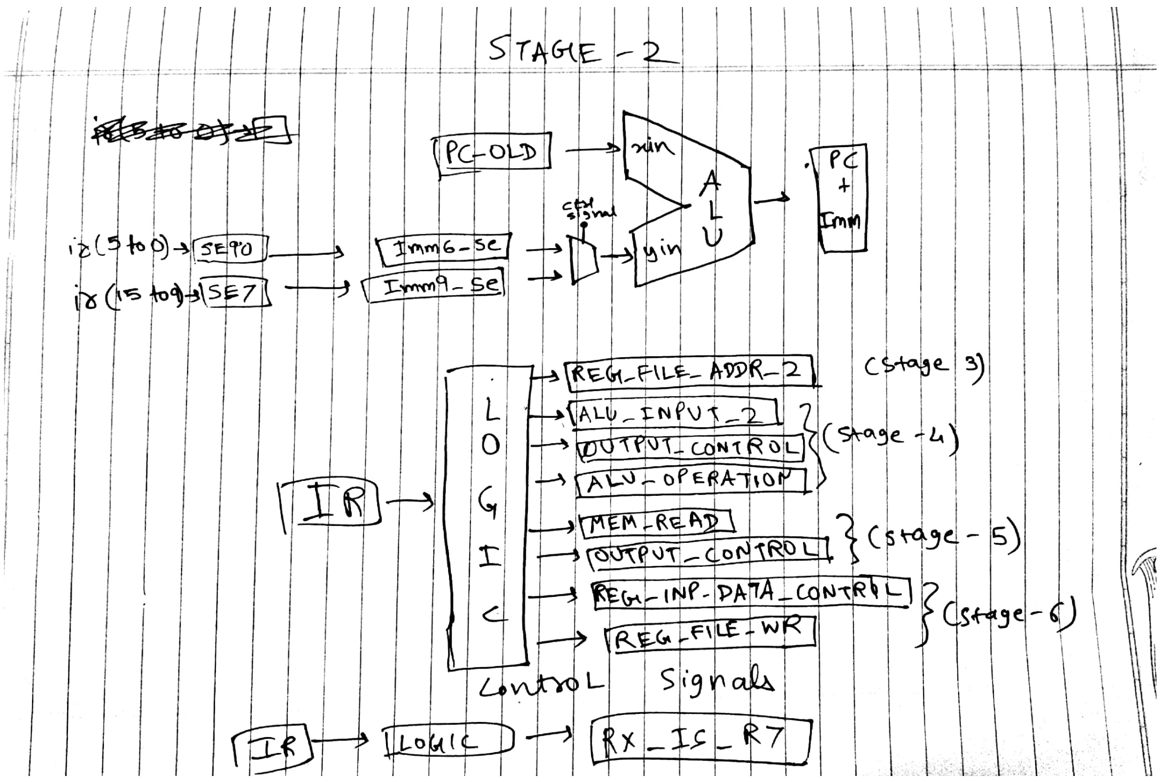
1. valid-in : Decides wheteher the current instruction is valid
2. pc-control : Decides whether to load PC+1 , value of register B or PC+Imm in the next instructions PC
3. reg-b-val : Forwards value of register B from memory to this stage
4. pc-plus-imm : PC+Imm value as input

2.1.2 Output

1. ir : the instruction to be executed
2. pc-old: The PC for current instruction
3. valid-out: Denotes whether current instruction is valid

Note: The outputs of this stage have been forwarded throughtout the pipeline and hence have not been rementioned in the other 5 stages.

2.2 Stage 2 - Instruction Decode



This is the most important stage in the pipeline. The results of this stage, in the form of control signals, control the other 5 stages in terms of their behaviour and hence their outputs. The results of this stage define the path that the pipeline will take from hereon.
The main aim of this stage is to decipher the type of instruction that has been loaded into the memory and hence modify the control signals for the following based on this. For ex: This stage decides whether the ALU in stage 4 will do the NAND operation or ADD.
Another major function that this stage defines is the detection of hazards in the pipeline. Thus, this stage de codes whether the inputs of the current instruction are in some form dependent on the outputs of the previous 3 instructions. This is then sent in form of bits to stage 3 so that it can read the forwaded values accordingly.
Apart from this , this stage also sends out bits for some specific instructions like BEQ, JAL,JLR , LM,SM that notify the stages that this is the curent instruction and hence they will react in the special way that is encoded in them.
This stage also contains one of its own ALU. The job of this ALU is to compute the PC+imm (sometimes 6 bit or sometimes 9-bit) and forward it too the the first stage along with a bit which decides whether this should be used as next PC or not.

The important inputs and outputs of this stage (apart from those of stage 1) are enumerated here:

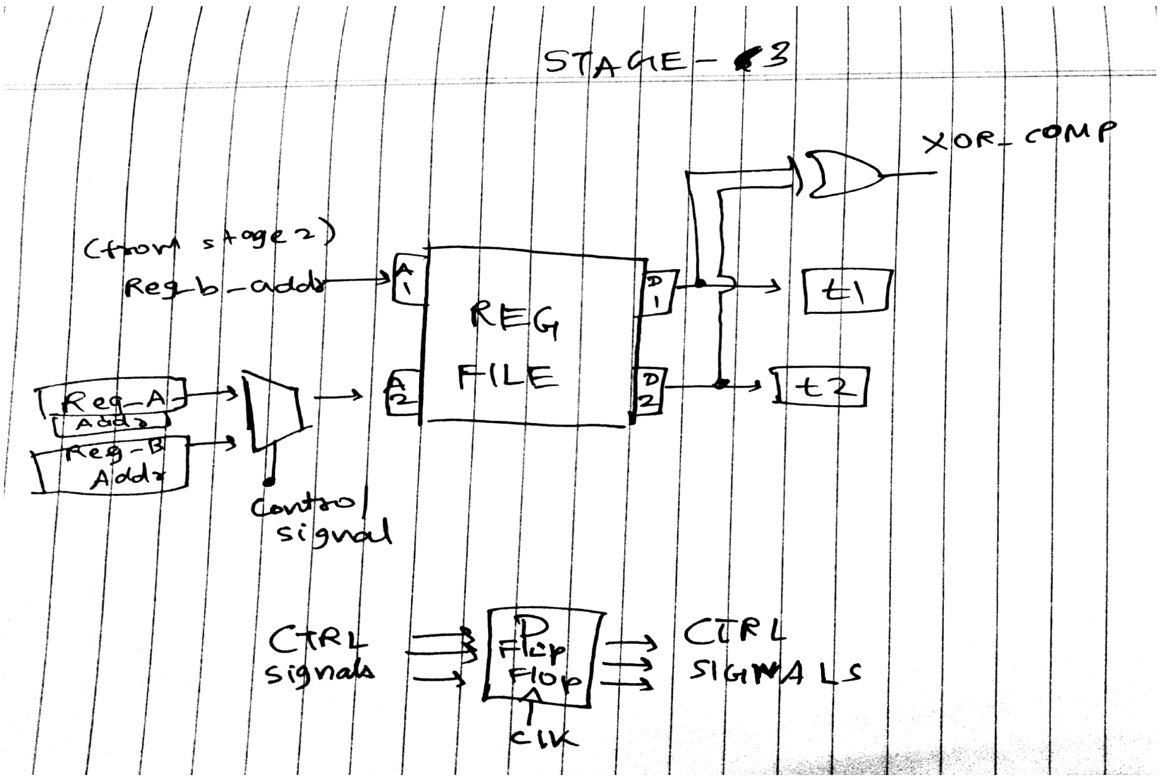
2.2.1 Inputs

1. pc-old-i : Incoming instruction's pc. Forwarded to next stage on clock edge.
2. valid-in : Shows if current instruction is valid
3. loadlukhi3 : Shows if current stage has to be halted as load followed by add type hazard has been detected and pipeline needs to be stalled.

2.2.2 Outputs

1. pc-plus-imm : PC+imm data to be sent to first stage
2. imm6,imm9,reg-x-addr : Direct decoding. Bits from the instruction which might include this kind of data is forwarded as it is. WHeteher the next stages use this or not is decided by the other control signal mentioned below.
3. reg-addr2-ctl-3: Bit that tells stage 3 whether the the operand to be fetched is at reg C address or Reg A address.The thing that is noted here that if Reg C is input to an instruction then Reg A cannot be an input to the instruction and vice-versa. Hence, this bit can be used without any issues.
4. mem-rd-5,reg-rd-6 : Tells stage 5 and 6 respectively whether memory or register is to be read or written to. In case of stage 6 only is write operation is needed or not. No read
5. read-from-a : If A is used as input
6. r-x-hzrd : One of these 3 bits is set if a hazard is detected in the current instruction. This be then tells stage 3 that the data has to be brought back from one of the later stages.This 3 bits are priority encoded and if two hazards are detected the data will be brought from the nearest stage to 3 ie most recent hazard.
7. rx-is-R7 : Bit to show is the r7 is used in instruction which is later used to keep R7 and PC as same

2.3 Stage 3 - Register/Operand Fetch



This stage of the pipeline is responsible to find and fetch the correct operands which will be used as input for stage 4.
Thus, almost all the hazards are resolved in this stage. If not resolved, this and all previous stages are halted as long as the hazard is resolved(ex: The load followed by add hazard)
Apart from this , this stage also has a XOR-circuit used for BEQ instruction. As the values of the registers are available in this stage, this is the earliest we can resolve the BEQ branch. The result of this XOR computation is the sent to the first two stages in form of valid bits which renders them invalid if branch is not taken and PC is also updated accordingly fro the first stage.
Thus, the operands and few of the control signals needed for stage 4,5,6 are forwarded ahead on encountering clock-edge.Th important input and output signals are:

2.3.1 Inputs

1. Outputs from stage 2 as signals.
2. rd-d1,rf-d2 : Address of the registers to be fetched

3. stagex-op: Outputs of stage 4,5 and 6. These are requires to resolve hazards as one of these will be forwarded as operands to next stage if hazard is detected.

4. r-x-hzrd : Shows which is the closest hazard detcted. Data is brought in from there.

2.3.2 Outputs

1. t1,t2 : The register values fetched from the registers or the further stages

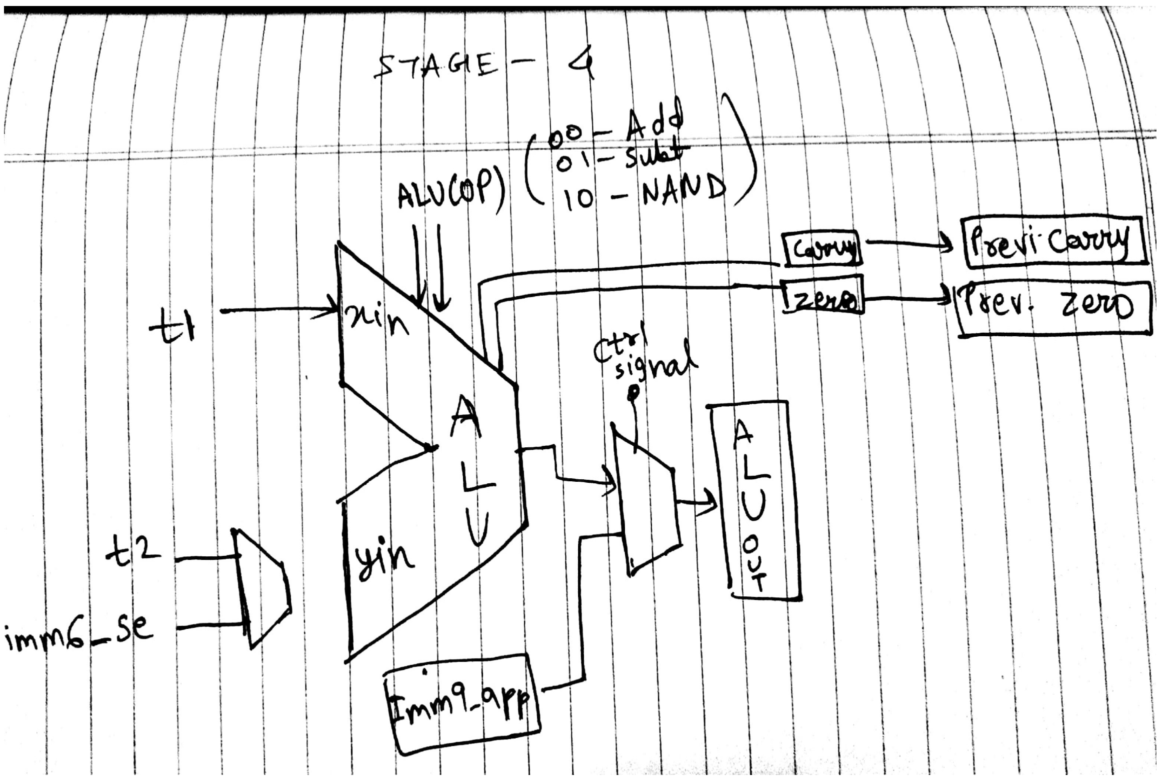
2. carry-yes,zero-yes : Bit is set if carry/zero is to be considered for this instruction. This bit is the only difference between normal ADD and ADC or ADZ.

3. xor-comp : Bit is set if the two register values fetched are same. If this bit is set the first two satges are rendered invalid and the pc value is updated. This mechanism resolves branches in the BEQ instruction.

4. load-hzrd-out-2x : Bit is set if load hazard is detected for register X.

5. rx-is-r7: Bit is set if register x is r7. Used for PC-R7 continuity.

2.4 Stage 4- Execute



The fourth stage is where the main 16-bit execution of the instruction takes place. It contains a 16-bit ALU with two inputs and its output is the one used in stage 5 and 6 to be written to the memory or register specified in the instruction.
This is also the place where the flags are modified and hence have two outputs zero and carry flags that are used by the following stages to make decisions whether to write back the data or not.
This stage was made to have two outputs , one a normal output from ALU or immediate data as per the instruction and another one for hazards. The hazard output contains the original PC value of this instruction in case of JAL and JLR instructions and the normal output otherwise.
The main inputs and outputs of this stage are:

2.4.1 Inputs

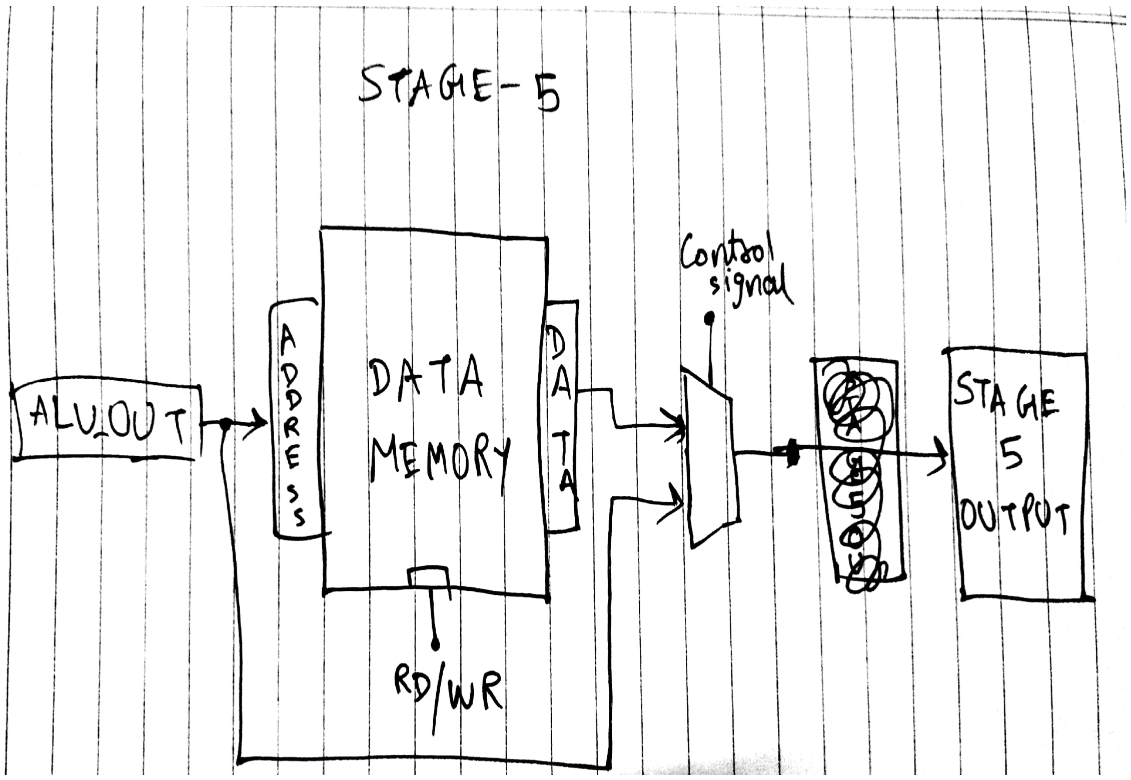
1. t1-in,t2-in : The two inputs to the ALU. t1 is always used as 1 input. t2 may be swapped out with some immediate data as per the instruction.
2. input-alu2-ctl : Defines which operation the ALU should perform. For ex.

’00’ for addition , ’01’ for Nand and so on.

2.4.2 Outputs

1. alu-out: Output of main ALU.
2. zero,carry : The two flags that will be used to to decide whether to write back data or not. These will be used by subsequent stages.
3. All the inputs from stage 3 not used by this stage are forwarded as they are om clock edge.

2.5 Stage 5:Memory Write/Read



This is the stage where all operations with data memory happens. Thus this is the latest stage where the instruction might receive operands. Hence, after this stage, no hazard resolution is required.
This stage has multiple inputs like t2-in ,alu-out that contain data and others like xxx-yes and others that contain control signals.
Since this is the only stage that deals with data memory, the data memory is considered as a subpart of this stage.
The outputs of this stage contain the values read from the memory and some of the outputs of previous stage , like the flags, that are required by the next stage.The main inputs and outputs are:

2.5.1 Inputs

1. alu-out-5: Output of the 5th stage. Used as memory address or as data to be written to memory. In some cases simply forwarded to 6th stage.
2. output-ctrl : Decides whether the input is to be forwarded as output or the data read from memory is to be forwarded as output.
3. t2-in: Value of t2 from previous stage. Is used by next stage in some cases.

Hence passed through this stage.

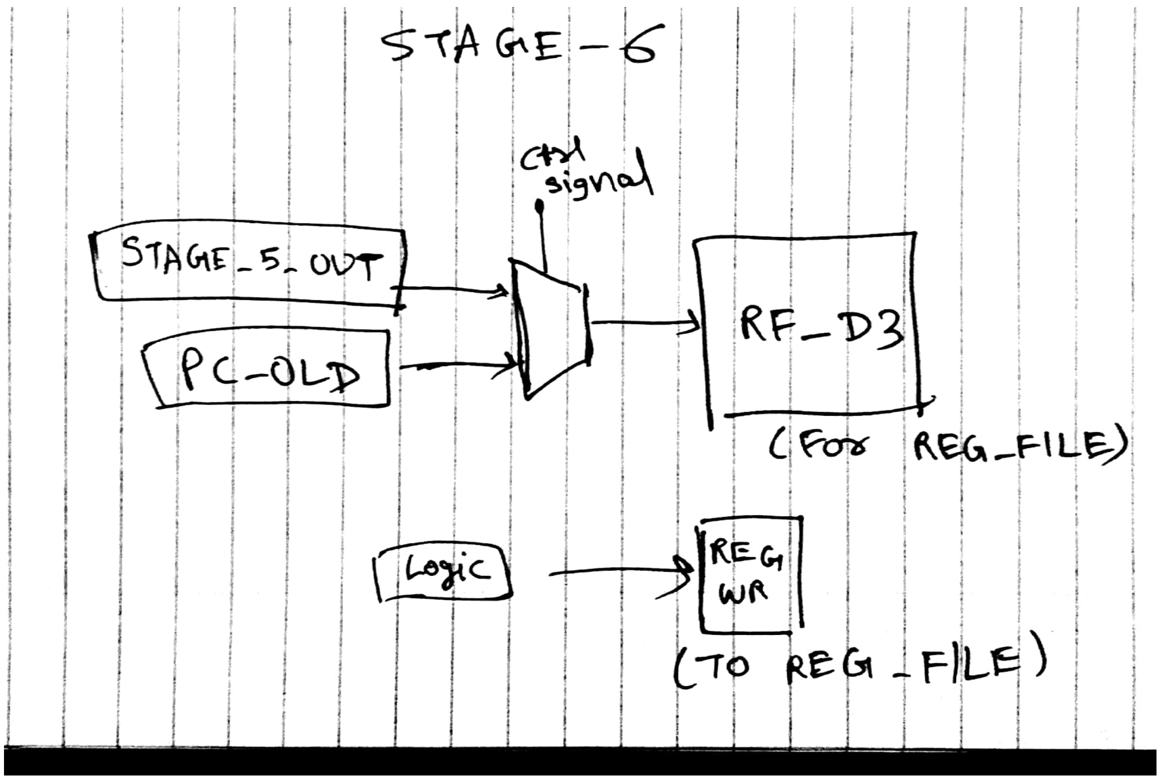
4. lm-active,sm-active: decide what is to be used as memory address. More explaination in lm-sm part.

2.5.2 Outputs

1. stage-5-out: Data read from memory or simply forwarded from previous stage.

2. Control signals from next stage forwarded as they are

2.6 Stage 6: Register Write



This is the last stage in the pipeline and has the least amount of logic associated with it.It has minimum number of outputs which mostly serve as inputs for stage 3 in case of hazards.
Some outputs like pc-to-r7 and wr7 have been specifically made for making pc and r7 as equal throughout the pipeline. This is modified and used when r7 is to written into.
The register file is used by multiple stages and hence is not a sub-part of this stage. Thus this stage has two main outputs one for the register number and the other for the data to be written in that register. The main inputs and outputs are listed below:

2.6.1 Inputs

1. lm-active,sm-active: Set if lm or sm is being executed currently.
2. p-carry-i,p-zero-i: The carry and zero flags.
3. pcarry-yes-i,carry-zero-i: Bits that decide whether the carry and zero flags have to be considered.

2.6.2 Outputs

1. rrf-d3: Decides which register is to be written into.
2. stage6-out: contains the data that has to be written into the given register
3. reg-wr-1: Bit that is set to 1 if the data is sent is actually to be written into the register. Zero is junk is transferred.
4. pc-to-r7,wr7: Contains PC and 1 if r7 is not to be written into as per the instruction. If r is destination of instruction, it is handled separately.

3 The LM and SM instructions

To execute these instructions, we made use of a special shifter register. When the LM or SM instruction arrived in the 4th stage the last eight bits of the instruction were written into the shifter register, and the clock signal to the first three stages was cut off.Each bit shifted out of the register in ascending order told us whether a particular register is to be loaded or stored (for LM and SM repectively).

We made a seperate finite state machine for executing these instructions which used the resources in stages 5 and 6 to execute the complete instruction in about 16 cycles.

4 Hazard Detection and Resolution

4.1 Arithmetic Instructions

- Instructions like ADD and NAND lead to the most basic hazards which can be resolved using simple forwarding
- Add instructions whose operands are destinations of the any of the previous three operands can be solved in this manner. If the same operand is dependent one more than 1 of the last 3 instructions, the data is fetched from the nearest stage i.e. the most recent instruction.
- Hazards like ADC followed by ADD are handled with the help of a valid bit. If the ADC instruction is not to be executed the data propagated from that instruction to later instructions is rendered invalid by the valid bit being 0. This is implemented by anding the valid bit with the hazard detection bit , and thus if ADC is not executed,hazard is not detected.
- Instructions that have R7 as input do not have any hazards as the current PC is directly used as operand instead of R7.
- Instructions with R7 as destination are handled separately. In such cases, the newer instructions in the pipeline will be rendered invalid and PC will be updated with R7.

4.2 Load Instruction

- Load instruction or LW executes in the 5th stage as all memory reads happen there.Thus,if load is followed by another instruction like add immediately whose input is output of load, the pipeline has to be stalled for one cycle to make the data available at stage 4.
- For second-level dependency, we first tried to treat it as a normal add-add 2nd level hazard. But due to anding of the clock to stall the pipeline, some instantaneous clicks were seen in RTL simulation. This issue was resolved by simple stalling the pipeline for two cycles.

4.3 Branch Instruction

- The input hazards for branch instructions were handled by the normal forwarding mechanism.
- The BEQ and JLR branch instructions were resolved in the third stage by making input valid bits of stage 1 and 2 zero and hence at most will result in two bubbles/stalls in the pipeline.
- The JAL instruction was resolved in the second stage of the pipeline itself by making valid bit of stage 1 zero and thus always results in only one bubble in the pipeline.

5 Disclaimer

All the work that has been presented and reported has been carried out by the members of this group without the help of any other batchmates or seniors.