CS 3410 P2 MINI-MIPS Processor Design Document

Oren Michaely (om72) and Pablo Llambias (pll438)

1. Overview

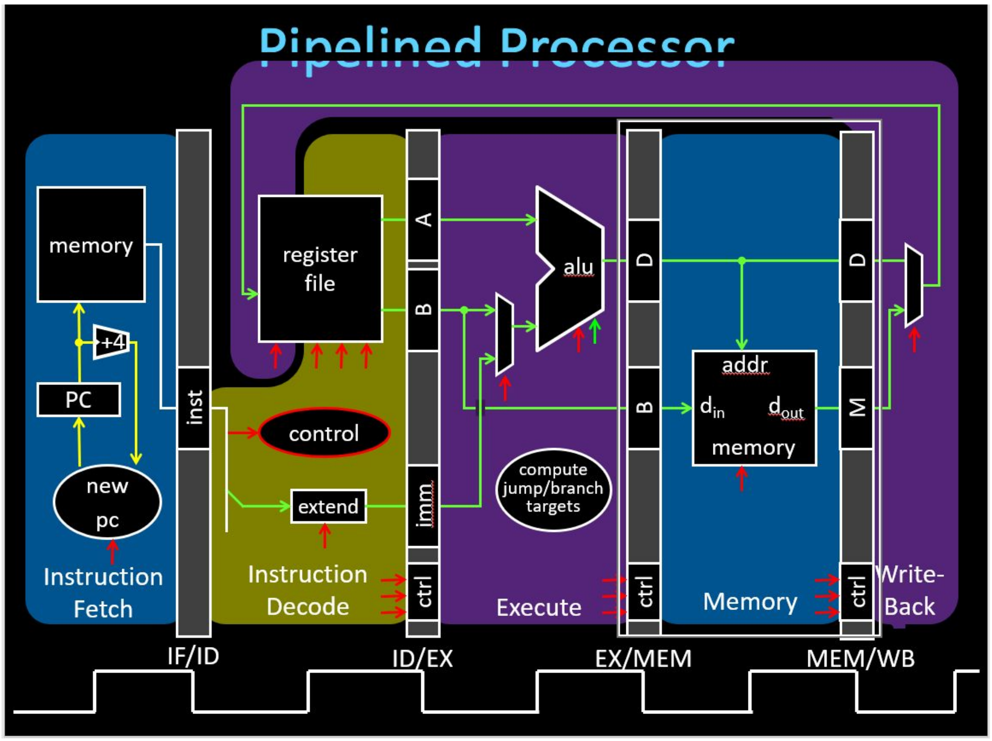
The purpose of the minMIPS is to design a processor which will perform a subset of the operations performed by the full MIPS processor. The purpose of the MIPS processor is to effectively and efficiently take in a 32-bit instruction and store the correct result to the correct register file, based on the given instruction. Additionally, it is important to make sure that this process of evaluating an instruction is done as efficiently as possible, as all operations done within the MIPS processor must be completed within a single clock-cycle. In order to maximize the efficiency, we will implement this MIPS processor using the pipelining paradigm. When implementing with pipelining, we will need to consider data hazards, specifically Ex hazards and Mem hazards. Different techniques will be used to avoid complications.

2. Component Design Documentation

2.1) MIPS processor

The minMIPS processor circuit will be split into five stages for pipelining. Additionally, control logic is implemented to avoid data hazards, as well as to decide which operation will be performed and which stage is accessed at each time.

2.1.1) Implementation Details



The MIPS Processor has 5 stages. The first stage is the Instruction Fetch stage in addition to fetching instructions, this first stage is in charge of determining if there are any hazards. The second stage is the Instruction Decode Stage, where the 32 bit instruction is “broken up”. The Third is the Execute stage, where the correct inputs are inputted into the ALU and the corresponding value is computed. The fourth stage is the Memory stage, which is in charge of storing the values in memory. In this project no values are stored to memory so this stage in the pipeline does nothing. The fifth and last stage is the Write-Back stage which writes back the output from the ALU to the correct register file. Each one of these steps and their implementation will be explained in further detail in the corresponding section of this design document.

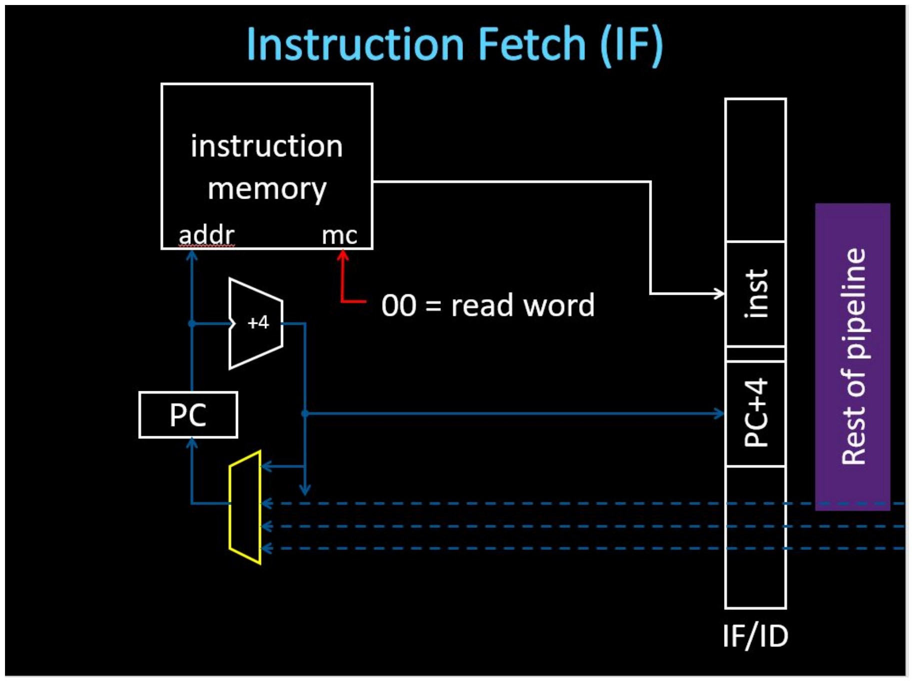
2.1.2) Evaluation

We will evaluate the efficiency of our MIPS processor once we build it. In general Pipelining improves the throughput by an amount proportional to the amount of stages in the pipeline.

2.2) Instruction Fetch

The PC calculates the new target address that the instruction is stored in in the ROM(Read-Only Memory). That instruction will be read and stored in a register in the IF/ID register file. The PC will then be incremented by 4 to prepare for the next instruction.

2.2.1) Implementation Details



In the Instruction fetch stage, we fetch a new instruction every cycle. Every cycle the PC points to the index of the instruction in the ROM memory. At the end of every clock cycle the PC is incremented by 4. We write the instruction bits(we will need it later for decoding) and the PC+4 (we will need it later for computing brack targets) to the pipeline register (IF/ID).

In order to increment the PC by 4, we will take the 30 most significant bits and pass them through the incrementer. Then, we will simply append the other two bits and have therefore incremented it by 4.

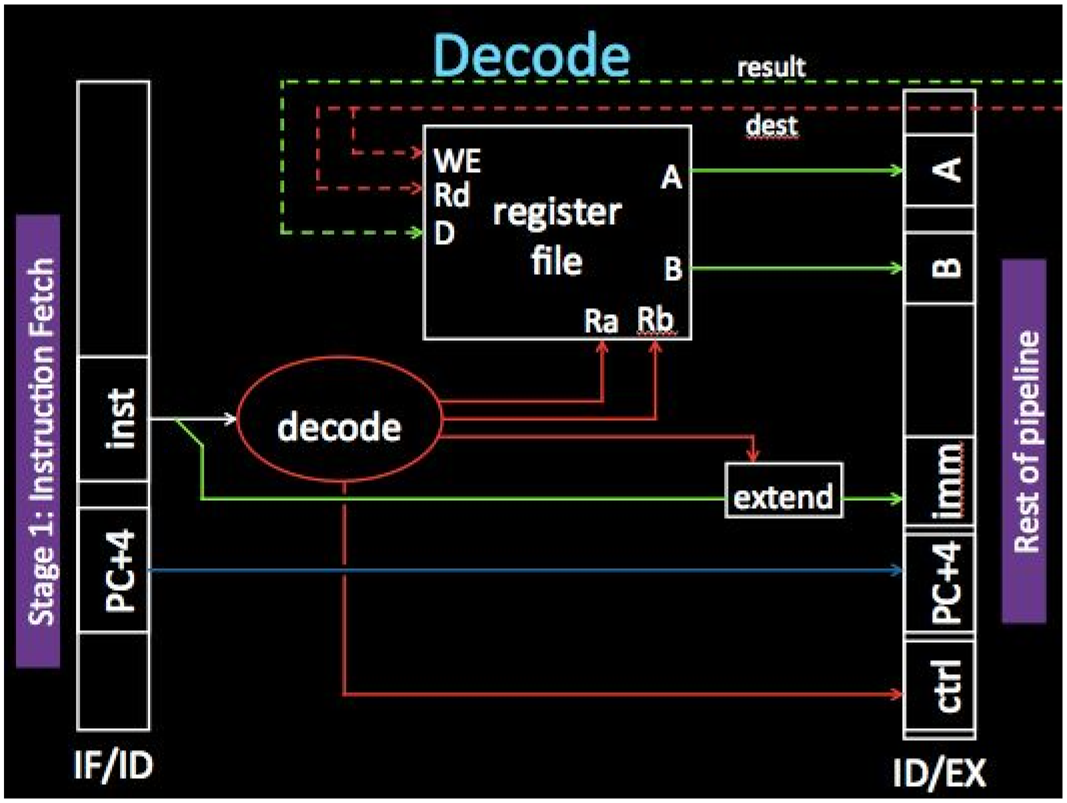
2.2.2) Evaluation

- Theoretically optimal, we will fill this in once we implement this stage.

2.3) Instruction Decode

The instruction decode stage is essential in the overall performance and correctness of the processor. It will take in an instruction, which is read form the IF/ID pipeline register, and will decode it, generating control signals that will determine which operation is performed in the execute state. It will pass these control signals to the ID/EX register. Additionally, it will pass controls to the Reg file, which determines whether A, B or both are read from the register file and passed to the ID/EX register file. Additionally, depending on the type of instruction, it will take the 16 bit immediate portion of the instruction from the IF/ID RF and zero-extend it into a 32-bit value that will be passed to a register in the ID/EX register file.

2.3.1) Implementation Details

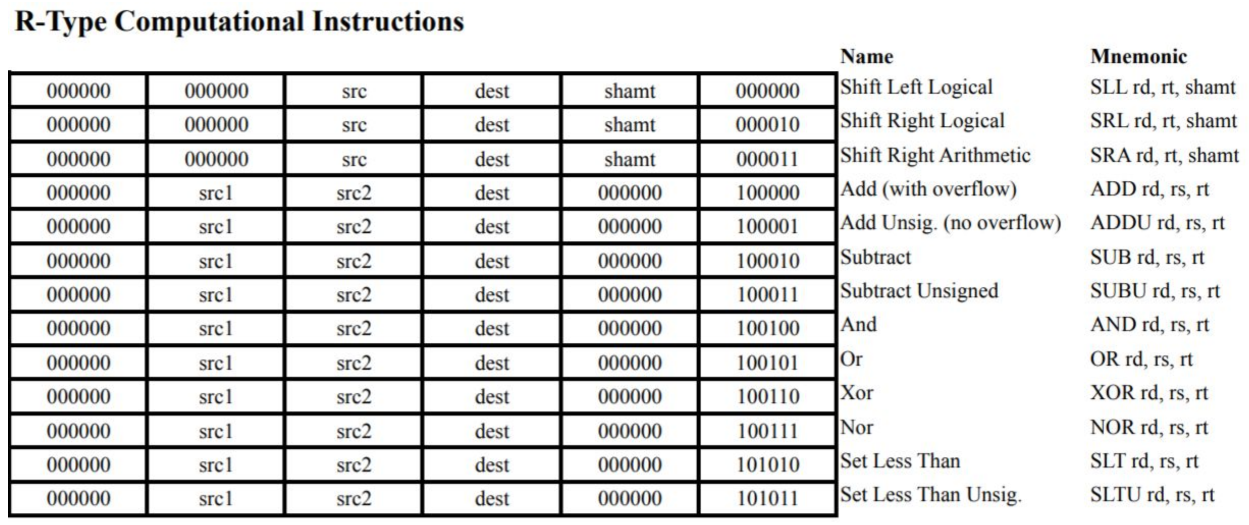


How to decode the instruction:

First, we will try to identify the type of the instruction (Between I/J-Type, R-Type and LW/SW). In order to do this we look at bits 26-31. If they are all 0’s we know that the instruction is R-Type.
If they are not all 0’s we know that we either have I/J-Type or LW/SW. To distinguish between these we look at bit 31. If bit 31 = 1 then we have LW/SW which we are not implementing this project so we have to make sure that the instruction goes through the pipeline and no changes are made to the register files .
The next step is distinguishing between I-Type and J-Type. In order to do this we look at bit 29. If bit 29=1 then we have I-Type, if not we have J-Type.

The next step is to decide what control signal will go into the the ctrl in the ID/EX register. At this point we know what type we have so we will do this type by type.

R-Type



R-Type Computational Instructions

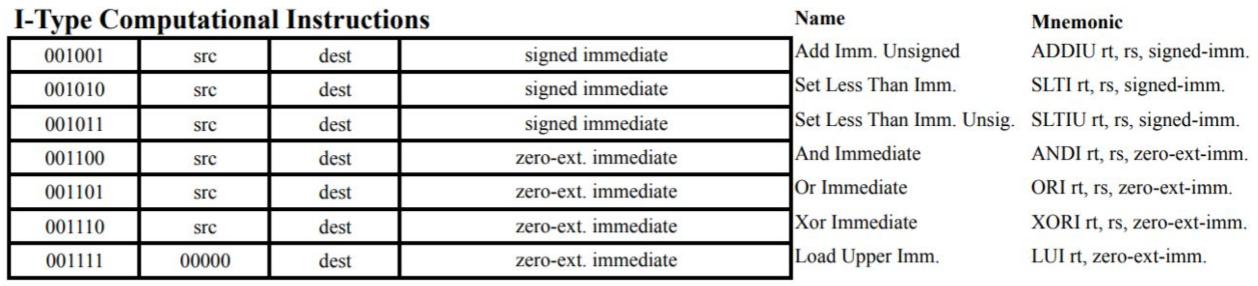
						Name	Mnemonic
000000	000000	src	dest	shamt	000000	Shift Left Logical	SLL rd, rt, shamt
000000	000000	src	dest	shamt	000010	Shift Right Logical	SRL rd, rt, shamt
000000	000000	src	dest	shamt	000011	Shift Right Arithmetic	SRA rd, rt, shamt
000000	src1	src2	dest	000000	100000	Add (with overflow)	ADD rd, rs, rt
000000	src1	src2	dest	000000	100001	Add Unsig. (no overflow)	ADDU rd, rs, rt
000000	src1	src2	dest	000000	100010	Subtract	SUB rd, rs, rt
000000	src1	src2	dest	000000	100011	Subtract Unsigned	SUBU rd, rs, rt
000000	src1	src2	dest	000000	100100	And	AND rd, rs, rt
000000	src1	src2	dest	000000	100101	Or	OR rd, rs, rt
000000	src1	src2	dest	000000	100110	Xor	XOR rd, rs, rt
000000	src1	src2	dest	000000	100111	Nor	NOR rd, rs, rt
000000	src1	src2	dest	000000	101010	Set Less Than	SLT rd, rs, rt
000000	src1	src2	dest	000000	101011	Set Less Than Unsig.	SLTU rd, rs, rt

The least significant 6 bits represent the function to be executed.

If the func bits most significant bit is a 0 we have some kind of shift operation (SLL,SRL,SRA,SLLV,SRLV,SRAV).
Now we have to decide if we have a shift by shift amount or by a variable amount. To do this we look at bit 2. If bit2=1 then we have to shift by a variable amount. Now we have SLLV, SRLV or SRAV. We have to look store the least significant 5 bits from the register that rs maps to into the shift amount in the ID/EX register. Then we need to shift the func code 2 bits to the right and the 2 bits to the left (not using the ALU) and then we input the least significant 4 bits into the opcode in the register.
If bit2=0 then we have SLL or SRL and we input the shift amount into the shift amount in the ID/EX register, and we shift our func bits left by one and input the least significant 4 bits into the opcode in the ID/EX.

If the func bits most significant bit is 1 we have an arithmetic operation (ADDU, SUBU, AND, OR, XOR, NOR, SLT, SLTU).
If bit 3 is 1 then we have SLT or SLTU, which both will not be inputted into the ALU, but rather into comparators, so a relevant value will be stored into the opcode in the ID/EX stage so that it does not enter the ALU in the next stage.
If bit 3 is 0, then we have (ADDU, SUBU, AND, OR, XOR, NOR). For these operations we will shift left by one, and input the least significant 4 bits into the opcode in the ID/EX register.

I-Type


I-Type Computational Instructions

				Name	Mnemonic
001001	src	dest	signed immediate	Add Imm. Unsigned	ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate	Set Less Than Imm.	SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate	Set Less Than Imm. Unsig.	SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate	And Immediate	ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate	Or Immediate	ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate	Xor Immediate	XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate	Load Upper Imm.	LUI rt, zero-ext-imm.

We can break this down into two subcircuits: One in which bit 28 = 0, the other in which bit 28 = 1.

If bit 28 = 0, we will consider I-type operations which have a signed immediate component (as opposed to a zero-ext immediate component). In this subcircuit, bit 0-15 will be concatenated with a bit extension of the signed bit of the immed input, namely bit 15. Thus bit 15 will also be fed into the the resulting bit 31 place, and this output will be passed to the ID/EX register file. Additionally, if bit 26 = 0, then we know it will be an ADDIU operation, so we will output bit 27, bit 26, bit 25 and a 0 into a 4-digit opcode that will be fed into the ID/EX pipeline register file. The 0 will come from bit 28 and will go into the least significant spot of the opcode. If instead bit 26 = 1, then we know it will either be a set less than immediate or set less than immediate unsigned. We will then pass an opcode containing bit 27, bit 26, bit 25 to the ID/EX pipeline register.

If bit 28 = 1, we will consider I-type operations which have a zero-ext immediate component (as opposed to a signed immediate component). In this subcircuit, bit 0-15 will be concatenated with a 16 bit zero-extension. The zero to be extended will come from the NOT of bit 28. The zero's will go into the 16 most significant bits, and the immed input will go into the 15 least significant bits. This output will be passed to the ID/EX register file. Additionally, if bit 25 and bit 26 both 1, then we know that the operation to be done is the Load Upper Immediate. We will thus pass an opcode into the output of this subcircuit that contains bit 25 and bit 26, signifying that within the zero-ext immed. Subcircuit, the operation to be done is an LUI. If they are not both 1,, then bit 27, 26, 25 and the not of bit 28 will be fed as a 4-bit opcode that will be fed into the ID/EX pipeline register file.

J-Type and Branches

For this project we do not need to implement these so we need to make sure that the instructions make it through the pipeline and no register values are changed.

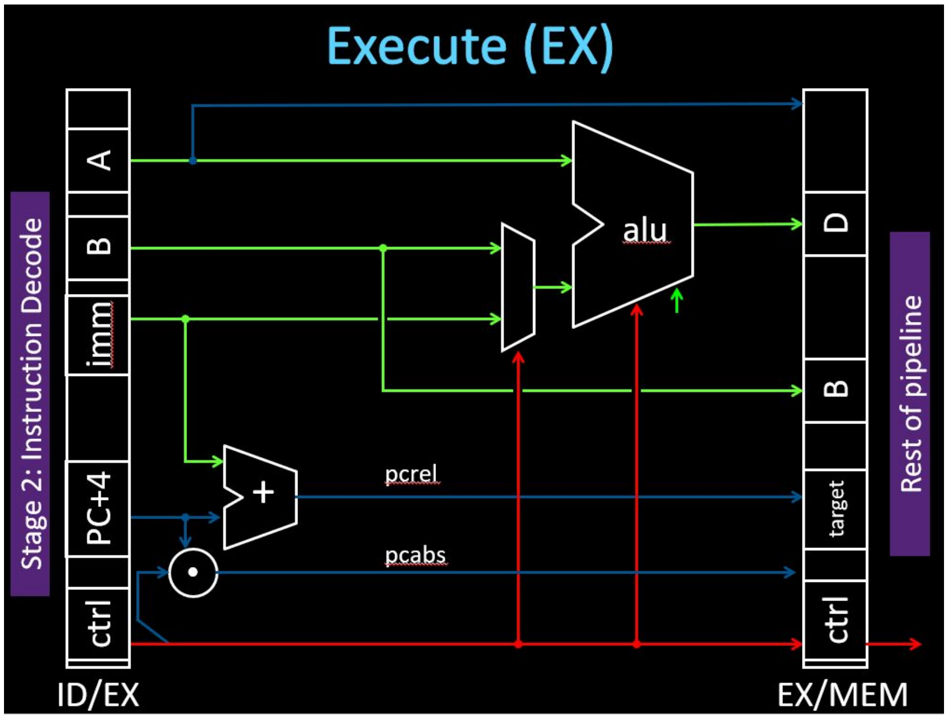
2.3.2) Evaluation

Nothing to evaluate at this point.

2.4) Execute

The execute stage feeds in the correct values to the ALU (from A and B or from A and imm), or if no ALU operation is needed.

2.4.1) Implementation Details



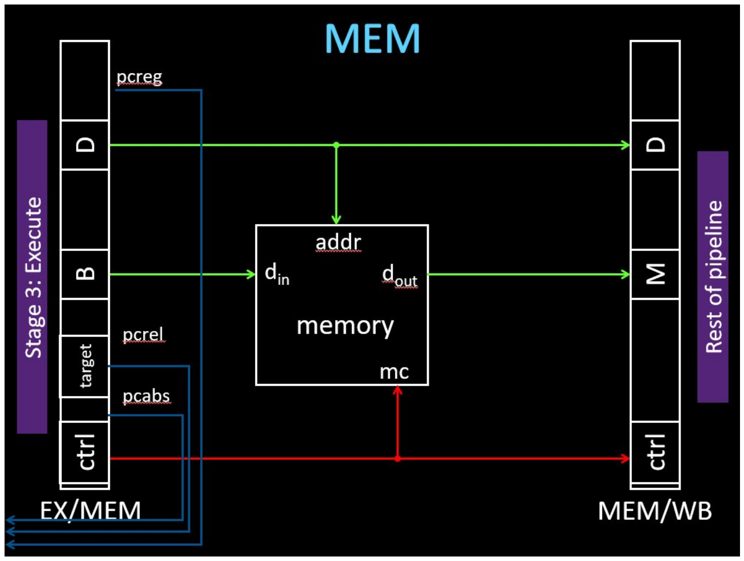
On every cycle we read the ID/EX pipeline register to get values and control bits, perform ALU operations and compute targets. In addition we write the relevant values to the EX/MEM pipeline register such as control information, and the result of the ALU operation. We do not need to store a memory location in this project. We also have a mux which determines whether a second register value is input into the ALU or whether our immediate output from the ID/EX pipeline register file is input. The control for this mux comes from one of the cntrl registers in the ID/EX pipeline register file.

2.4.2) Evaluation (when implemented)

2.5) Memory

Is not Implemented in this project, need to pass through it without changing any register values

2.5.1) Implementation Details

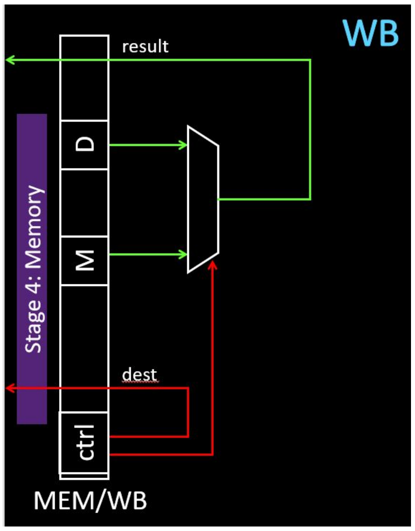


We need to pass the output of the ALU to the MEM/WB pipeline register, pass the control information to the MEM/WB pipeline register as well as the register address which we would like to store the output to.

2.6)Write Back

Read MEM/WB pipeline register to get the values and control bits select and write the value to the register file.

2.7.1) Implementation Details



Write back to correct register based on the ctrl.

2.7.2) Evaluation

...

2.7) Forwarding

One of the ways to avoid hazards. We will implement the hazard control in the MINI-MIPS 32 circuit using the following conditional statements. In general the purpose of forwarding is to avoid stalling. We forward results from the Execute stage or Writeback stage back into the ALU to be more efficient.

2.8.1) Implementation Details

Ex Hazard

-
  ```
  if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd
  == ID/EX.RegisterRs)) ForwardA = 10
  if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd
  == ID/EX.RegisterRt)) ForwardB = 10
  ```

MEM Hazard

-
  ```
  if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
      and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 01

  if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
      and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 01
  ```
- Short explanation as well

2.8.2) Evaluation

….