Design Documentation

Samuel Thomas (sgt43) Elisha Sword (eds88)

September 26, 2017

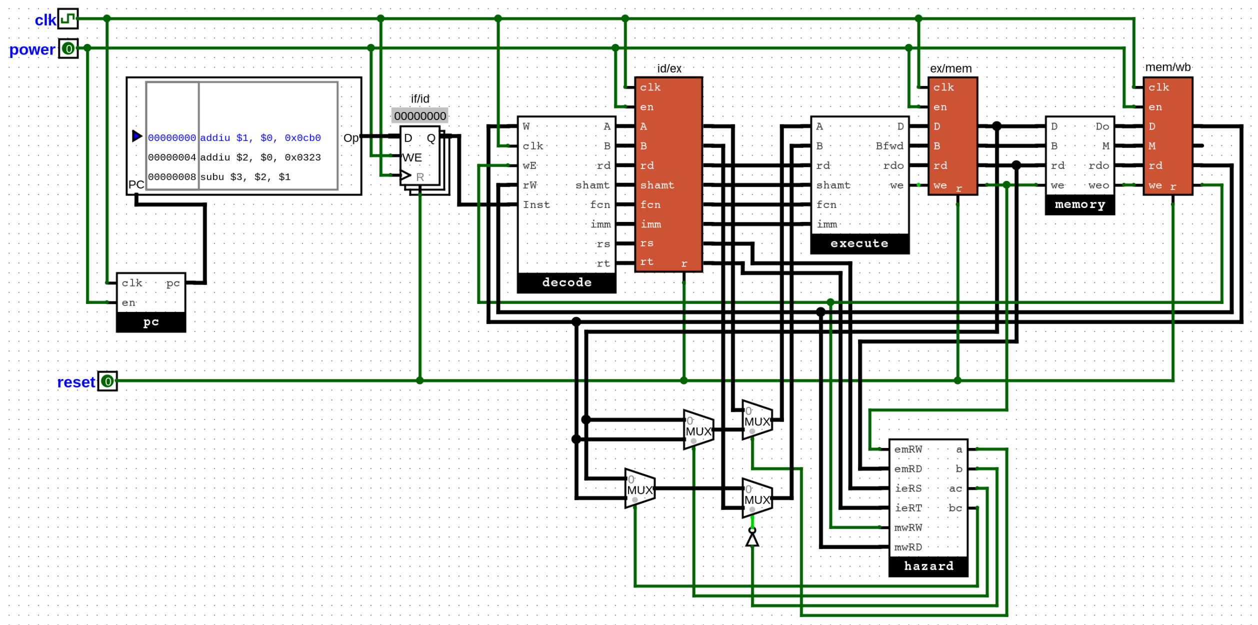
1 Introduction

The purpose of this document is to communicate the design of our 5-stage pipelined MIPS processor. We go into detail of the design of each of the five stages in the pipeline as well as the pipeline registers that connect the stages. This document assumes knowledge of MIPS and of logic gates. It is not meant to be instructive in these regards.

A pipelined processor is a processor that executes each instruction by passing it through different stages. At the end of each stage, the results are stored. At the beginning of each stage, the stage grabs the output from the previous stages. Although pipelining increases the number of clock cycles that it takes to execute each instruction, we can reduce the clock speed and increase the throughput of our processor.

2 Overview

We present a 5-staged pipeline design. Instructions are fetched in the first stage. Instructions are decoded and registers are read in the second stage. The third stage executes an instruction. The fourth stage writes values to memory if required. The fifth stage writes the results of the execute stage back into the register file.

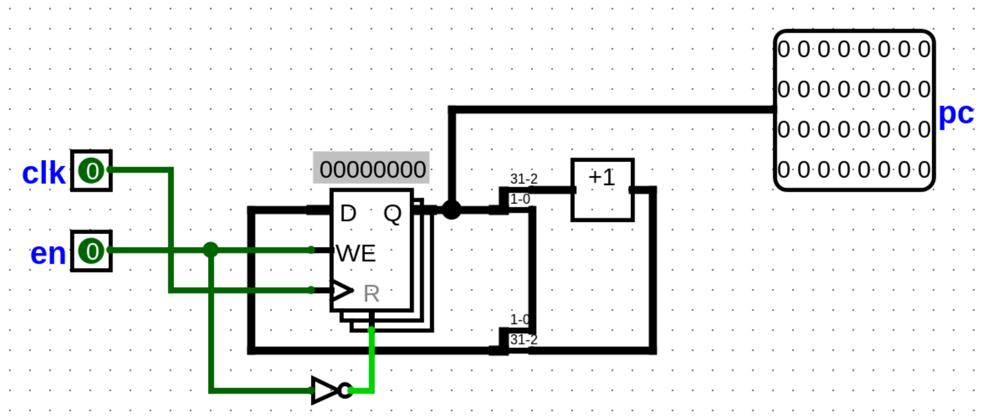


```
00000000 addiu $1, $0, 0x0cb0
00000004 addiu $2, $0, 0x0323
00000008 subu $3, $2, $1
```

3 Instruction Fetch Stage

This stage fetches the next instruction to execute. We store all of the instructions in Program ROM and keep track of where we are in the execution with `PC`. Every clock cycle the `PC` is incremented.

3.1 Circuit Diagram



3.1.1 PC Incrementer

We store the `PC` in a 32-bit register. Every clock cycle, we increment the upper 30 bits by 1 with a `+1 Incrementer` without touching the lower 2 bits. This gives us the desired effect of advancing the program counter by four every clock cycle. The `en` input is an on/off switch. When turned off, it resets the counter.

3.1.2 Latch

We will store the fetched instruction in the latch at the end of this stage. We are not supporting jump instructions so we do not store `PC+4`.

3.2 Correctness Constraints

Fetch the next instruction on the rising edge of the clock cycle.

- Correctly increment `PC` by 4 every clock cycle.

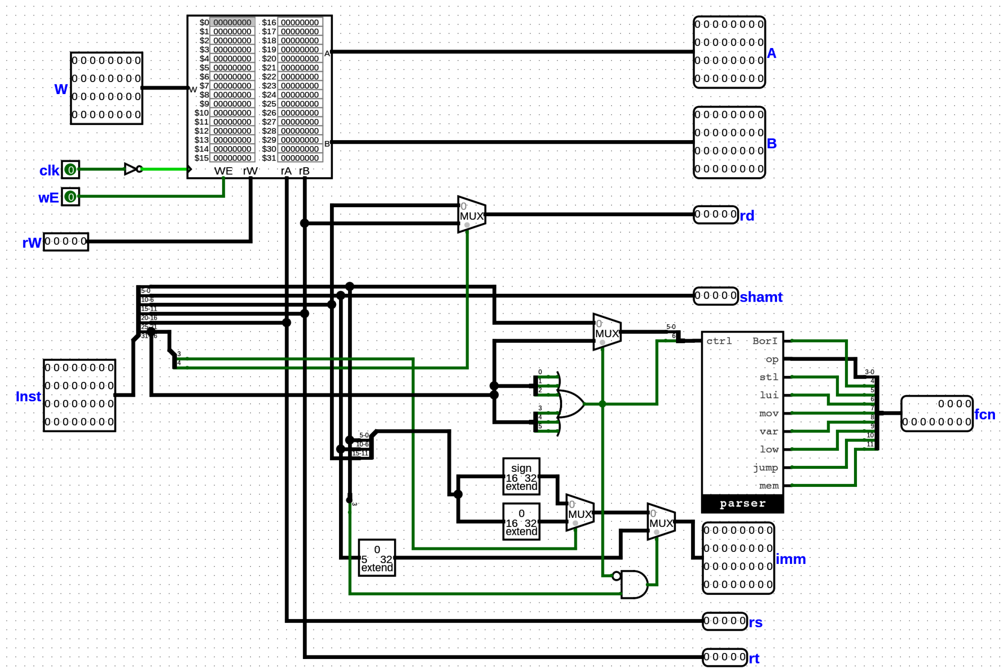
3.3 Testing

Test that the instructions are fetched in the correct order.

4 Instruction Decoding Stage

This is the stage where our register file lives. In the first half of the clock cycle, we write input `W` to register `rW` if `wE` is high. In the second half of the clock cycle, we read registers `rA` and registers `rB` and output them as `A` and `B` respectively. We also split the opcode into `rd`, `shamt`, `fcn`, and `imm` so that future stages simpler. `Fcn` is determined from a `7 bit` control signal that is passed into `parser`. More on this below.

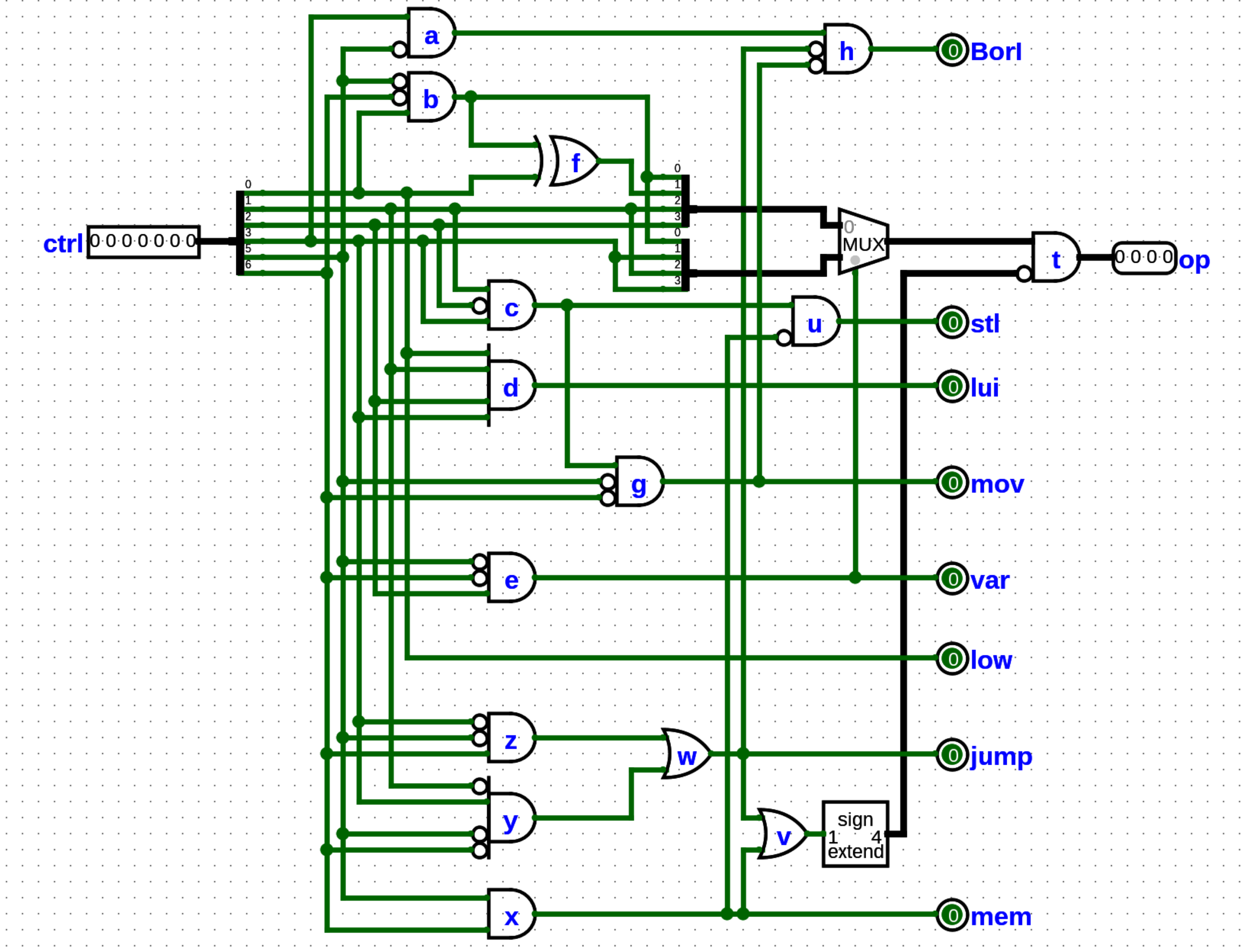
4.1 Circuit Diagram



4.1.1 Outputs

- `A` and `B` are read from the register file based on `rA` (bits 21-25) and `rB` (bits 16-20).
- The output `rd` is determined by `bit 3` of the opcode. When this bit is low, `rd = bits 11-15`. When this bit is high, `rd = bits 16-20`.
- `shamt` is always equal to `bits 6-10`.
- `fcn` is a 12 bit number that will be used in the execute stage to determine what operation to give to the ALU. It is encoded as the following:
  - `bits 0-3` are the ALU opcode.
  - `bit 4` chooses between B and Imm.
  - `bit 5` is high when we are executing a STL instruction.
  - `bit 6` is high when we are executing a LUI instruction.
  - `bit 7` is high when we are executing a MOV instruction.
  - `bit 8` is high when we are executing variable shift instructions.
  - `bit 9` is the lowest bit of the control input.
  - `bit 10` is high when we are executing jump instructions.
  - `bit 11` is high when we are executing mem instructions.
- `imm` is `bits 0-15` (either sign extended or zero extended depending on `bit 2` of the `fcn` part of the instruction), or `bits 6-10` zero extended. This is determined by `bit 3` of the opcode. The purpose of this is to have a 32-bit zero to compare against in the execute stage.
- `rs` is simply `bits 21-25` forwarded onto the next stage for determining hazards.
- `rt` is simply `bits 16-20` forwarded onto the next stage for determining hazards.

4.2 Parser



This is where the bulk of the decoding takes place. The input is a `7 bit` control signal. The lower 6 bits of this signal come from either the opcode, or the funct portion of R-type instructions. The highest bit of this signal is 0 is the opcode of the instruction is all zeros and 1 otherwise. We use this bit to differentiate between between control signals that would otherwise be identical. For example, MOVN and SLTIU would otherwise have identical control signals.

The circuit logic is so complicated that it is difficult to describe in words. Instead, a table of the possible inputs to corresponding outputs is in the appendix. We will explain what each output is used for in the Execute Stage.

4.3 Correctness Constraints

- `W` is correctly written to `rW` in the first half of the stage.
- `A` and `B` are correctly read from the registers based on `rA` and `rB`.
- `rD` is correctly chosen based on the instruction.
- `shamt` is fetched correctly from the instruction.

- `fcn` correctly chooses between the opcode and the special function portion of the instruction and is correctly parsed.
- `imm` correctly outputs sign-extended lower 16-bits, zero-extended lower 16-bits, and 32-bit zero.

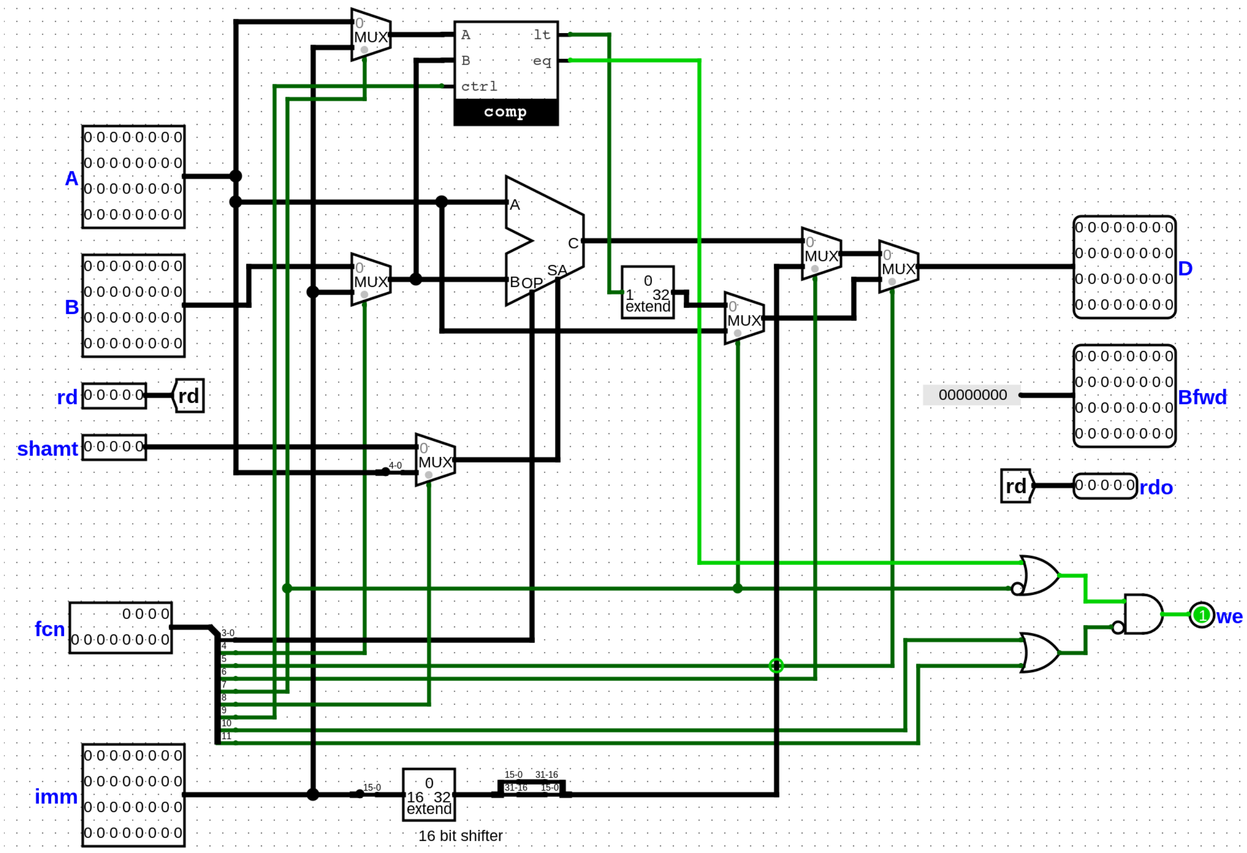
4.4 Testing

The `parser` circuit is tested for correctness by using the table in the appendix as a Test Vector in Logism. The whole decode stage is correct if programs execute on the entire processor correctly.

5 Execute Stage

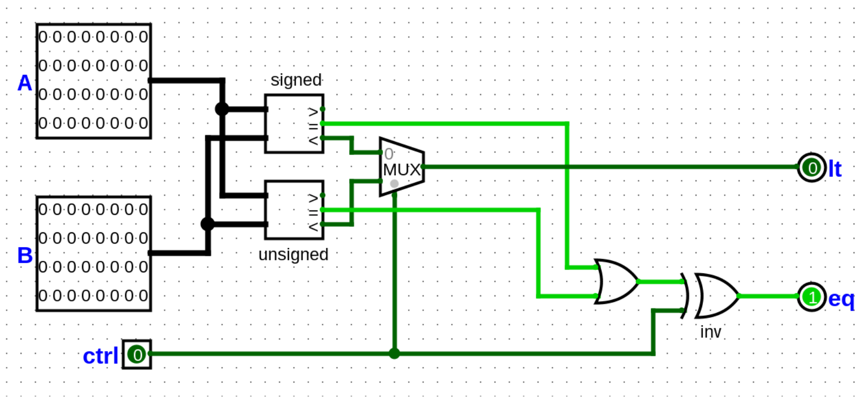
This stage executes all of the instructions that we will implement in this project.

5.1 Circuit Diagram



There are two subcircuits (`comp` and `ALU`). `Comp` performs comparison operations for SLT, SLTU, SLTI, SLTIU, MOVN, and MOVZ. `ALU` performs all of the other arithmetic operations.

5.1.1 Comp



This circuit uses an unsigned and signed comparator to perform $A < B$. We also use the comparators to perform $A = B$. An `XOR` gate is used to invert this when `ctrl` is high. `Ctrl` also switches between signed and unsigned comparators.

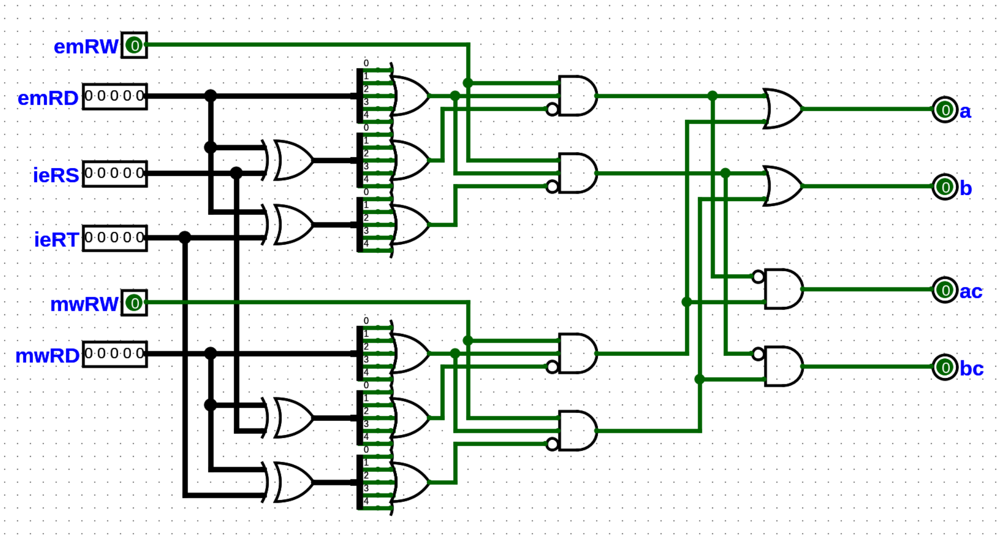
5.2 Correctness Constraints

- `we` should be on for all instructions except when MOVN or MOVZ return false.
- Addition, Subtraction, Shifting, Logic operations, Comparison operations should all execute as expected.
- Any instructions not yet implemented, (memory, branch, and jump instructions).

5.3 Testing

We wrote a python program to randomly generate Test Vectors for every possible input (including memory, branch, and jump instructions). We manually tested edge cases.

5.4 Hazards



This subcircuit holds all the hazard detection logic. Output `a` and `b` correspond to whether there is a data hazard for `A` or `B` (these are the values read from the register file). Output `ac` and `bc` are high if there is a MEM hazard on `A` or `B` respectively. They are low if there is a EX hazard.

These outputs are simply fed into muxes that replace the `A` or `B` input with values from the EX/MEM stage for EX hazards and from the MEM/WB stage for MEM hazards.

We wrote a series of programs that test all possible hazards to ensure that this logic is correct.

6 Memory Stage

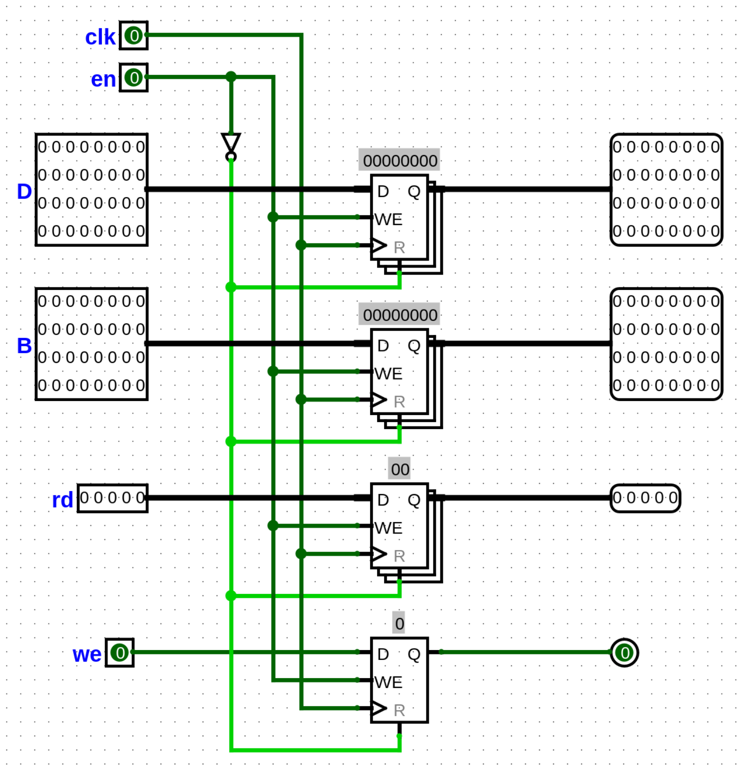
In this project, this stage doesn't do anything. The only reason that we include it here is in preparation for the next project. It simply forwards it's inputs to the next stage.

7 Write-Back Stage

For this project, there is no logic in this stage. It simply pipes `D`, `rd`, and `we` back to the decode subcircuit where it is written to the register file. To deal with MEM hazards, we pull down `rd` into the hazard logic. This is discussed in greater detail in the hazard section.

8 Pipeline Registers

In between each stage, there is a pipeline register. These simply contain a register for each input that are written to on the rising edge of the clock. Below is the implementation of the execute-memory pipeline register. All of the others are implemented in a similar fashion.



9 Summary

In this document, we presented a design for a limited 5-stage pipelined MIPS processor. Such a design greatly increases the performance of our processor by allowing multiple instructions to be executed simultaneously.

10 Appendix

Parser Opcode Table

instr	**ctrl[7]**	**BorI**	**op[4]**	**stl**	**lui**	**mov**	**var**	**low**	**jump**	**mem**
addiu	1001001	1	001x	0	0	0	0	1	0	0
andi	1001100	1	1000	0	0	0	0	0	0	0
ori	1001101	1	1010	0	0	0	0	1	0	0
xori	1001110	1	1100	0	0	0	0	0	0	0
slti	1001010	1	xxxx	1	0	0	0	0	0	0
sltiu	1001011	1	xxxx	1	0	0	0	1	0	0
addu	0100001	0	001x	0	0	0	0	1	0	0
subu	0100011	0	011x	0	0	0	0	1	0	0
and	0100100	0	1000	0	0	0	0	0	0	0
or	0100101	0	1010	0	0	0	0	1	0	0
xor	0100110	0	1100	0	0	0	0	0	0	0
nor	0100111	0	1110	0	0	0	0	1	0	0
slt	0101010	0	xxxx	1	0	0	0	0	0	0
sltu	0101011	0	xxxx	1	0	0	0	1	0	0
movn	0001011	0	xxxx	1	0	1	0	1	0	0
movz	0001010	0	xxxx	1	0	1	0	0	0	0
sll	0000000	0	000x	0	0	0	0	0	0	0
srl	0000010	0	0100	0	0	0	0	0	0	0
sra	0000011	0	0101	0	0	0	0	1	0	0
sllv	0000100	0	000x	0	0	0	1	0	0	0
srlv	0000110	0	0100	0	0	0	1	0	0	0
srav	0000111	0	0101	0	0	0	1	1	0	0
lui	1001111	1	xxxx	0	1	0	0	1	0	0
j	1000010	0	0000	0	0	0	0	0	1	0
jr	0001000	0	0000	0	0	0	0	0	1	0
jal	1000011	0	0000	0	0	0	0	1	1	0
jalr	0001001	0	0000	0	0	0	0	1	1	0
beq	1000100	0	0000	0	0	0	0	0	1	0
bne	1000101	0	0000	0	0	0	0	1	1	0
blez	1000110	0	0000	0	0	0	0	0	1	0
bgtz	1000111	0	0000	0	0	0	0	1	1	0
bltz	1000001	0	0000	0	0	0	0	1	1	0
bgez	1000001	0	0000	0	0	0	0	1	1	0
lw	1100011	0	0000	0	0	0	0	1	0	1
lb	1100000	0	0000	0	0	0	0	0	0	1
lbu	1100100	0	0000	0	0	0	0	0	0	1
sw	1101011	0	0000	0	0	0	0	1	0	1
sb	1101000	0	0000	0	0	0	0	0	0	1