# ELEC3608: Single Cycle Processor Part 1

Lawrence Wakefield

12/09/15

## Contents

# 1 Introduction

For the assignment we have been given a partially complete implementation of a single cycle processor. Below is a list of the instructions that have already been implemented.

- addi
- addiu
- ori
- lui
- lw
- sw
- addu
- subu
- sltu
- or

We have also been given a mips assembly program `fib.s` which computes the fibbonaci sequence. The processor provided does not yet implement all the instructions required for `fib.s` to run correctly. A list of the instructions which need to be implemented are as follows:

- sll
- jal
- slt
- bne
- be
- jr
- j

The goal of this assignment is to understand the current design of the single cycle processor, and then modify the design to implement the required instructions.

# 2 Datapath

Below is the modified datapath of the single cycle processor. Control signals are red, some have not been connected to the control unit, this is for legibility reasons. Similarly the `rs` and `imm16` signals going into the multiplexer toward the bottom of the multiplexer come from the register file (`rs` out) and the sign extended imm 16 unit, respectively. Also note that there are only three ports exposed on the processor, `boot`, `isWr` and `wrData`. From these ports we can program the device and then boot it to start the program to run.

The modifications made to the datapath are further detailed below.

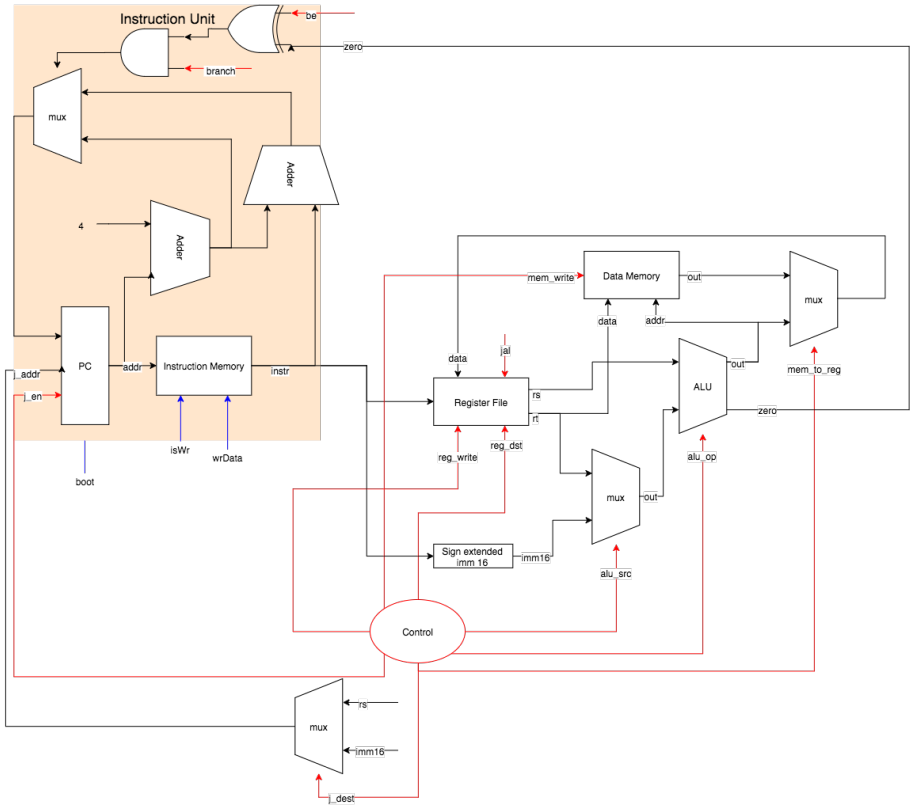


Figure 1: Modified datapath of a single cycle processor

## 2.1 New control signals

There was an addition of five new control signale including `jal`, `j_dest`, `be`, `branch` and `j_en`.

### 2.1.1 jal

`jal` is a signal to the register file that indicates that register $31 should be set to the return address, for use with a subroutine call.

### 2.1.2 j_dest

`j_dest` is a signal which determines where a jump will be made from, it is the selector for a multiplexer of either register $rs or the imm16 value.

### 2.1.3 be

`be` is a signal which determines whether the branch condition is on equal (`beq`) or on not equal (`bnq`).

### 2.1.4 branch

`branch` is a signal which indicates whether we are executing a branch instruction or not.

### 2.1.5 j_en

`j_en` is a signal which indicates whether we will be executing a jump instruction or not.

## 2.2 New internal signals

There was an addition of three new internal signals including zero, j_addr and pc_addr.

### 2.2.1 zero

The `zero` signal is a flag signal on the output of the ALU used to show whether the outcome of the last arithmetic operation was zero or not.

#### 2.2.2 j_addr

`j_addr` is the signal selected with `j_dest` that holds the address where the processor will jump to.

#### 2.2.3 pc_addr

`pc_addr` holds the instruction of the next address that the processor will execute. This address is selected based upon the branch logic.

### 2.3 New units

The main modifiations to the processor relate to the instruction unit. There has been addition of two multiplexers, one adder, one `xor` gate and one `and` gate. The two gates and multiplexer make up the branching logic. If the branching logic determines that the next address will be a branching address, then the new adder increments the program counter to the branch address. The final multiplexer is used to determine whether the processing is jumping to an imm16 address or an address contained in a register.

## 3 Truth table

Below is the truth table of the control signals that are set based upon the instruction.

| instr | reg_dst | alu_src | alu_op | mem_to_reg | reg_write | mem_write | branch | beq | j_en | j_dest | jal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| addi | 0 | 1 | add | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| addiu | 0 | 1 | addu | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| ori | 0 | 1 | or | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| lui | 0 | 1 | add | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| lw | 0 | 1 | add | 1 | 1 | 0 | 0 | x | 0 | x | 0 |
| sw | x | 1 | add | x | 0 | 1 | 0 | x | 0 | x | 0 |
| addu | 1 | 0 | addu | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| subu | 1 | 0 | subu | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| sltu | 1 | 0 | sltu | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| or | 1 | 0 | or | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| sll | 1 | 0 | sll | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| jal | x | x | x | 0 | 0 | 0 | 0 | x | 1 | 0 | 1 |
| slt | 1 | 0 | slt | 0 | 1 | 0 | 0 | x | 0 | x | 0 |
| bne | x | 0 | subu | 0 | 0 | 0 | 1 | 0 | 0 | x | 0 |
| beq | x | 0 | subu | 0 | 0 | 0 | 1 | 1 | 0 | x | 0 |
| jr | x | x | x | 0 | 0 | 0 | 0 | x | 0 | 1 | 0 |
| j | x | x | x | 0 | 0 | 0 | 0 | x | 1 | 0 | 0 |

An brief overview of the signals in the truth table is outlined below.

### 3.1 Immediate instructions

Immediate instructions include addi, addiu, ori, lui

- reg_dst is set to 0 as we want the values to be saved in register $t
- alu_src is set to 1 to select the imm16 value
- alu_op is dependent on the instruction
- mem_to_reg is set to 0, we don't want to write data from the memory to a register
- reg_write is set to 1, we want to save the output to a register
- mem_write is set to 0, we aren't writing to memory
- branch, j_en and jal are set to zero, we aren't jumping or branching
- beq, j_dest don't matter

### 3.2 Arithmetic instructions

Arithmetic instructions include addu, subu, sltu, or, sll and slt

- reg_dst is set to 1 as we want the values to be saved in register $d
- alu_src is set to 0 to select the value in register $t
- alu_op is dependent on the instruction
- mem_to_reg is set to 0, we don't want to write data from the memory to a register
- reg_write is set to 1, we want to save the output to a register
- mem_write is set to 0, we aren't writing to memory
- branch, j_en and jal are set to zero, we aren't jumping or branching
- beq, j_dest don't matter

### 3.3 Load/store instructions

Load and store instructions include lw and sw

- reg_dst
  - is set to 0 for load as we want the values to be saved in register $t
  - doesn't matter for store
- alu_src is set to 1 to select the imm16 value
- alu_op is set to add
- mem_to_reg

  - is set to 1 for load, we are loading into a register
  - doesn't matter for save
- reg_write
  - is set to 1 for load, we are writing to a register
  - is set to 0 for save, we are not writing to a register
- mem_write
  - is set to 0 for load, we are not writing to memory
  - is set to 1, we are writing to memory
- branch, j_en and jal are set to zero, we aren't jumping or branching
- beq, j_dest don't matter

### 3.4 Jump instructions

Jump instructions include j, jr and jal

- reg_dst doesn't matter
- alu_src doesn't matter
- alu_op doesn't matter
- mem_to_reg is set to 0, we don't want to write data from the memory to a register
- reg_write is set to 0, we don't want to write to a register
- mem_write is set to 0, we aren't writing to memory
- branch is set to 0, we aren't branching
- beq doesn't matter
- j_en is set to 1, we will be jumping
- j_dest
  - is set to 1 for jr, we want to jump to the val in a register
  - is set to 0 for j and jal, we want to jump to an immediate value
- jal
  - set to 1 for jal, we want to link the return address
  - set to 0 for j and jr, we don't want to link the return address

### 3.5 Branch instructions

Branch instructions include beq and bnq

- reg_dst doesn't matter
- alu_src is set to 0 to select the value in register $t

- alu_op is set to subu as we want to compare values by looking at the zero flag
- mem_to_reg is set to 0, we don't want to write data from the memory to a register
- reg_write is set to 0, we don't want to write to a register
- mem_write is set to 0, we aren't writing to memory
- branch is set to 1, we are executing a branch instruction
- j_en and jal are set to 0, we aren't jumping
- j_dest doesn't matter