**Design Document**

**Register Availability**

* 16, 16-bit general purpose registers
* $0, constant zero register
* $ra, return address for storing the return address from a procedure call
* $sp, stack pointer for storing the address of the current position in the stack
* $v0, return register for storing the return value from a procedure call
* $a0, $a1, argument register for storing arguments for the procedure call
* $k0, OS kernel for interrupt management
* $t0-$t3, temporary registers for storing temporary values that can be overwritten through procedure calls
* $s0-$s4, save registers for storing important values that won’t be overwritten through procedure calls

**Instruction Descriptions**

* ADD - Add rs, rt, rd (R-Type)
  + Adds rs and rt and stores the result in rd
* ADD IMMEDIATE - Addi rd, Imm (A-Type)
  + Adds rd and Imm and stores the result back in rd
* AND - And rs, rt, rd (R-Type)
  + Logically And’s rs and rt and stores the result in rd
* BRANCH EQUAL - Beq rs, rt, rd (R-Type)
  + If rs is equal to rt, then sets PC equal to the memory address in rd
* BRANCH NOT EQUAL - Bne rs, rt, rd (R-Type)
  + If rs is not equal to rt, then sets PC equal to the memory address in rd
* JUMP - J Imm (J-Type)
  + Jumps to address in Imm which has been shifted left once and concatenated the first three bits of the PC to the most significant bit.
* JUMP AND LINK - Jal Imm (J-Type)
  + Jump to address in Imm and sets $ra to the PC value
* JUMP REGISTER - Jr rd, Imm (A-Type)
  + Jump to address in the rd register
* LOAD WORD - Lw rs, rd, Imm (I-Type)
  + Loads the immediate at address rs into rd
* OR - Or rs, rt, rd (R-Type)
  + Logically Or’s rs and rt and stores the result in rd
* SET LESS THAN - Slt rs, rt, rd (R-Type)
  + If rs is less than rt, rd equals 1, otherwise rd equals 0
* SHIFT LEFT LOGICAL - Sll rs, rd, Imm (I-Type)
  + rs is shifted left by the Imm value, and is stored in rd
* SHIFT RIGHT LOGICAL - Srl rs, rd, Imm (I-Type)
  + rs is shifted right by the Imm value, and is stored in rd
* SHIFT RIGHT ARITHMETIC - Sra rs, rd, Imm (I-Type)
  + rs is shifted right arithmetically by the Imm value, and is stored in rd
* SUBTRACT - Sub rs, rt, rd (R-Type)
  + Subtract rt from rs and stores the result in rd
* STORE WORD - Sw rs, rd, Imm (I-Type)
  + Stores the value rs at the 4 bit Imm offset of rd

**Register Descriptions**

|  |  |  |
| --- | --- | --- |
| Register Name | Number | Usage |
| $0 | 0 | Constant zero register |
| $ra | 1 | Return address (used by function call) |
| $sp | 2 | Stack pointer |
| $v0 | 3 | Return value of a function |
| $a0 | 4 | Argument 1 for procedure call |
| $a1 | 5 | Argument 2 for procedure call |
| $k0 | 6 | Reserved for interrupt OS kernel |
| $t0 | 7 | Temporary (not preserved across procedure call) |
| $t1 | 8 | Temporary (not preserved across procedure call) |
| $t2 | 9 | Temporary (not preserved across procedure call) |
| $t3 | 10 | Temporary (not preserved across procedure call) |
| $s0 | 11 | Saved (preserved across procedure call) |
| $s1 | 12 | Saved (preserved across procedure call) |
| $s2 | 13 | Saved (preserved across procedure call) |
| $s3 | 14 | Saved (preserved across procedure call) |
| $s4 | 15 | Saved (preserved across procedure call) |

**Instruction Format Descriptions**

* R-Type

|  |  |  |  |
| --- | --- | --- | --- |
| OP (4 bits) | rs (4 bits) | rt (4 bits) | rd (4 bits) |

* + R-Type instructions receive two source registers (rs and rt) and store the result in the destination register (rd)
* I-Type

|  |  |  |  |
| --- | --- | --- | --- |
| OP (4 bits) | rs (4 bits) | rd (4 bits) | Imm (4 bits) |

* + I-Type instructions receive one source register (rs) and one immediate value, and stores the result in the destination register (rd)
* J-Type

|  |  |
| --- | --- |
| OP (4 bits) | Imm (12 bit) |

* + J-Type instructions use the 12 bit immediate value as an address to jump to
* A-Type

|  |  |  |
| --- | --- | --- |
| OP (4 bits) | rd (4 bits) | Imm (8 bits) |

* + A-Type Instructions receive an 8 bit immediate value and stores the result in the destination register (rd). Sign extension of one 8 bit immediate then adding another 8 bit immediate value will yield a 16 bit value.

**Machine Language Translation**

* add $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 0000 | 0111 | 1000 | 1001 |

* addi $t0, 0000 0001

|  |  |  |
| --- | --- | --- |
| 0001 | 0111 | 0000 0001 |

* and $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 0010 | 0111 | 1000 | 1001 |

* beq $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 0011 | 0111 | 1000 | 1001 |

* bne $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 0100 | 0111 | 1000 | 1001 |

* j 0000 0000 0001

|  |  |
| --- | --- |
| 0101 | 0000 0000 0001 |

* jal 0000 0000 0001

|  |  |
| --- | --- |
| 0110 | 0000 0000 0001 |

* jr $t0, 0000 0001

|  |  |  |
| --- | --- | --- |
| 0111 | 0111 | 0000 0001 |

* lw $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1000 | 0111 | 1000 | 0001 |

* or $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 1001 | 0111 | 1000 | 1001 |

* slt $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 1010 | 0111 | 1000 | 1001 |

* sll $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1011 | 0111 | 1000 | 0001 |

* srl $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1100 | 0111 | 1000 | 0001 |

* sra $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1101 | 0111 | 1000 | 0001 |

* sub $t0, $t1, $t2

|  |  |  |  |
| --- | --- | --- | --- |
| 1110 | 0111 | 1000 | 1001 |

* sw $t0, $t1, 0001

|  |  |  |  |
| --- | --- | --- | --- |
| 1111 | 0111 | 1000 | 0001 |

**Example Assembly Language Programs**

# v0 = n

addi $sp, -12

sw $s0, 0($sp)

sw $s1, 1($sp)

sw $ra, 2($sp)

and $s0, $0, $0 #Load Blank

ori $s0, 2 #put value of 2 in (m)

or $s1, $0, $a0 #move to saved register

Loop:

or $a0, $s1, $0 #n

or $a1, $s0, $0 #m

jal gcd #call gcd

addi $v0, -1 #Subtract by one. Don’t care about #under(over)flow due to just want to see if the #value had been 1. Ends up as gcd(n,m) - 1 != 0

beq $v0, $0, 2 #go past the jump

addi $s0, 1 #Increment by 1

j Loop

EndLoop:

lw $s0, 0($sp)

lw $s1, 1($sp)

lw $ra, 2($sp)

addi $sp 12

jr $ra

gcd:

and $t0, $0, $0 #Force Zero

addi $t0, 2 #Load 2, Used as value of commands to skip

bne $v0, $0, $t0 #skip not equal (don’t return)

or $v0, $0, $a1 #Move b to return

jr $ra #Return

addi $t0, 4 #Currently has 2 so skip end lines minus 2, Used #as commands to skip for while loop (total 6)

and $t2, $0, $0 #Set t2 to zero

addi $t2, 2 #Amount to skip to for if inside the while

begin:

beq $a1, $0, $t0 #If b != 0 jump t0 forward

slt $a0, $a1, $t1 #Set t1 to 1 if a less then b

bne $t1, $0, $t2 #Jump to else (forward 2)

sub $a0, $a0, $a1

j begin #End of If, Start of Else

sub $a1, $a1, $a0

j begin #Back up to look check

or $v0, $0, $a0 #Move a to return

jr $ra #Return

**Assembly Language Fragments**

* Loading an address into a register
  + addi $t0, 0000 0001 #Adds 0000 0001 to $t0  
    sll $t0, $t0, 1111 #Shifts left by 16 bits  
    addi $t0, 1000 0000 #Adds the lower 8 bits
* Iterations
  + loop :  
     beq $t0, $t1, $t2

addi $t0, 0000 0001 #increment 1

j loop #continue loop

* Conditional Statements
  + loop :

beq $t0, $t1, $t2 #If t0 is equal to t1, go to value in $t2 (1000 1000 1000)

addi $t0, 0000 0001 #increment 1

j loop #continue loop

value of $t2 :

jr $ra, 0000 0000 # goes to return address before the loop

* Reading and writing to display register
  + lw $t0, $t1, 0001

sw $t0, $t1, 0001

**Machine Language Translations of Example Programs**

* Conditional Statements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0011 | 0111 | | 1000 | 1001 |
| 0001 | 0111 | | 0000 0001 | |
| 0101 | 0000 0000 0001 | |
| 0111 | 0001 | 0000 0000 |

* Reading and writing to display register

|  |  |  |  |
| --- | --- | --- | --- |
| 1000 | 0111 | 1000 | 0001 |
| 1111 | 0111 | 1000 | 0001 |

* Loading an address into a register

|  |  |  |  |
| --- | --- | --- | --- |
| 0001 | 0111 | 0000 0001 | 0001 |

* Iteration

|  |  |  |
| --- | --- | --- |
| 0001 | 0111 | 0000 0001 |
| 0101 | 0000 0000 0001 |

**Milestone Two**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | A-Type | R-Type | lw/sw | Beq/bne | j/jal | I-Type |
| Inst Fetch | IR[PC]  PC=PC+2 | | | | | |
| Inst Decode  Reg Fetch | A = Reg[IR[11-8]]  B = Reg[IR[7-4]]  C = Reg[IR[3-0]] | | | | | |
| Execution Address Comparison  Beq/bne/j done | ALUout = A op IR[7-0] | ALUout = A op B | ALUout = A + IR[3-0] | ALUOut = PC +C | PC = PC[15-13] || IR[11-0] << 1  $ra = PC | A= B op IR[3-0] |
| Mem access  I,A,R Done | Reg[IR[11-8]] = ALUout | Reg[IR[3-0]] = ALUout | MDR = Mem[ALUout]  Mem[ALUout] = A | If ( A==B)  PC= ALUout |  | Reg[IR[7-4]] = B |
| Lw done |  |  | Reg[IR[7-4]] = MDR |  |  |  |

Components:

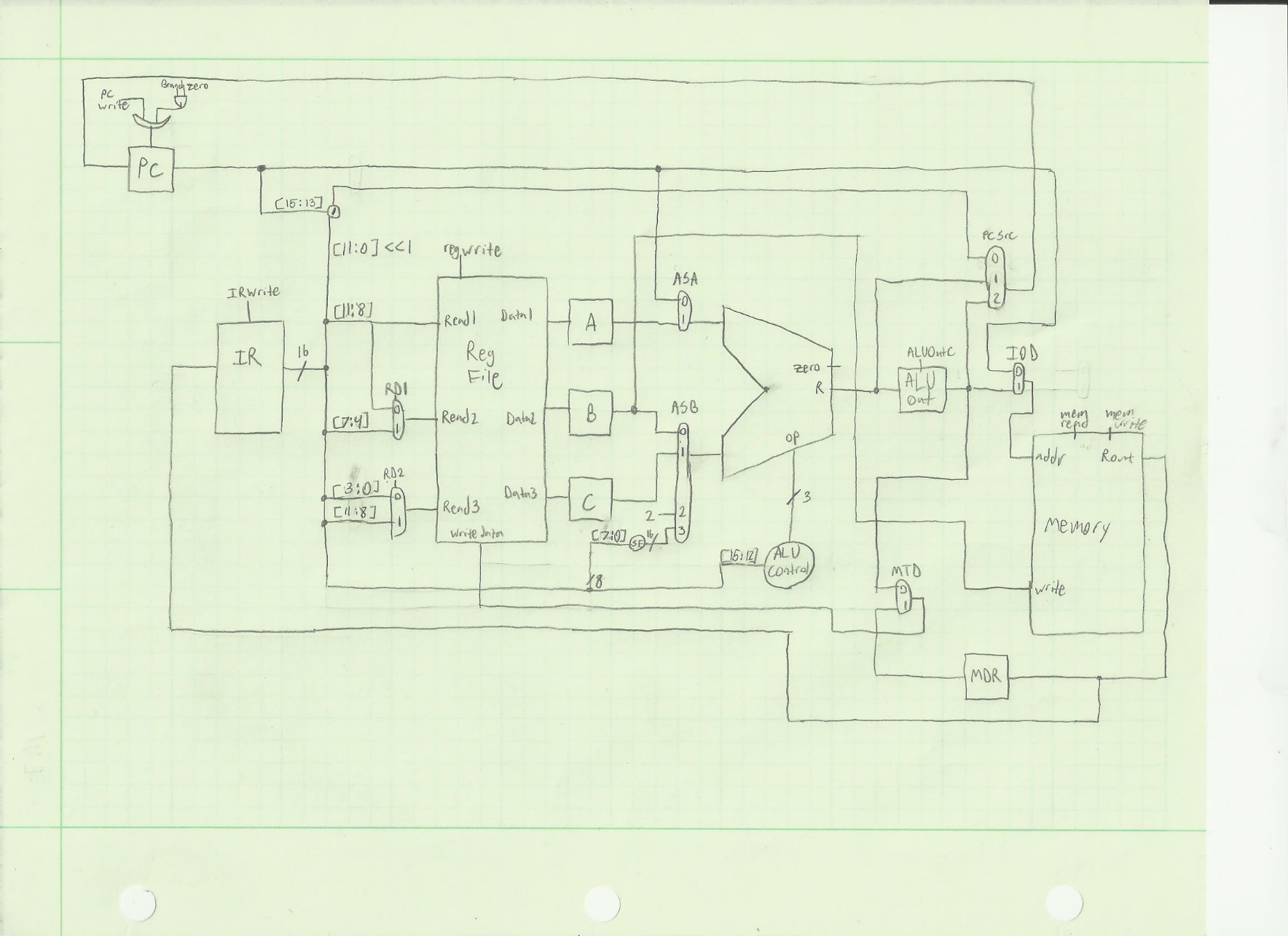
All considered to use 16 bits and transfer in 16 bit segments unless stated otherwise.

1. Program Counter Register
   1. Stores the Current Instruction Address.
   2. Inputs – Next Address
   3. Outputs – Current Address
   4. Controls – PCWrite, Zero, PCWriteCond
2. Memory
   1. RAM that contains the instructions and data.
   2. Inputs – Address, Data to Write
   3. Outputs – Data / Instruction
   4. Controls – MemRead, MemWrite
3. Instruction Register
   1. Stores the Current Instruction
   2. Inputs – Instruction from Memory
   3. Outputs – Current Instruction
   4. Controls – IRWrite
4. Register File
   1. Contains 16, 16 Bit Registers
   2. Inputs – Read Register 1,2 Read/Write Register 3, Write Data
   3. Outputs, Read Data 1,2,3
   4. Controls – RegWrite
5. A Register
   1. Stores the Data from Read Data 1 of the Register File
   2. Inputs – Read Data 1
   3. Outputs – Data in Register
   4. Controls - None
6. B Register
   1. Stores the Data from Read Data 2 of the Register File
   2. Inputs – Read Data 2
   3. Outputs – Data in Register
   4. Controls - None
7. C Register
   1. Stores the Data from Read Data 3 of the Register File
   2. Inputs – Read Data 3
   3. Outputs – Data in Register
   4. Controls - None
8. ALU
   1. Performs arithmetic and logical operations
   2. Inputs – ALUsrcA, ALUsrcB
   3. Outputs – zero, overflow, ALUout
   4. Controls – ALU Control
9. ALUout Register
   1. Stores the value from the ALU
   2. Inputs – Data from ALU
   3. Outputs – Data in register
   4. Controls - None
10. Memory Data Register (MDR)
    1. Stores the value from memory
    2. Inputs – Data from Memory
    3. Outputs – Data in register
    4. Controls – None
11. ALU Control
    1. Determines the ALU Control Code from the ALUop Code
    2. Inputs – Instruction[15-13]
    3. Outputs – ALUop Code
    4. Controls – AluOutControl
12. MemtoReg Mux
    1. Switches between the ALUout and MDR Registers
    2. Inputs - MDR, ALUout
    3. Outputs – Data to Write to register
    4. Controls MemToReg
13. IorD Mux
    1. Determines the Address piped into memory
    2. Inputs – PC, ALUout
    3. Outputs – Address
    4. Controls – IorD
14. RegDst1 Mux
    1. Determines the part of the instruction to pipe into Read 2 of the Register File
    2. Inputs – IR[11-8], IR [7-4]
    3. Outputs – One of the Inputs into Register Write Data
    4. Controls – RegDst1
15. RegDst2 Mux
    1. Determines the part of the instruction to pipe into Read 3 of the Register File
    2. Inputs – IR[11-8], IR[7-4], IR[3-0]
    3. Outputs – One of the Inputs into ALU Data 1
    4. Controls – RegDst2
16. ALUSrcA Mux
    1. Determines the data to pipe into ALU Data 1
    2. Inputs - A, PC
    3. Outputs – One of the Inputs
    4. Controls – ALUSrcA
17. ALUSrcB Mux
    1. Determines the data to pipe into ALU Data 2
    2. Inputs B, Sign Extended IR[3-0], Sign Extended Shifted Left 1 IR[3-0]
    3. Outputs – One of the Inputs into ALU Data 2
    4. Control – ALUSrcB
18. PCSource Mux
    1. Determines the Address to be put into the PC
    2. Inputs, ALU Data Out, ALUOut, PC[15-13] || IR[11-0] << 1
    3. Outputs – The address to be put into the PC
    4. Control – PCSource

Instruction Testing

1. ADD
   1. Test Basic Cases
      1. 1 + 1 = 2
      2. 1 + -1 = 0
      3. -1 + -1 = -2
   2. Test Overflow
      1. 0xFFFF + 0x1 =
2. ADDI
   1. Test Basic Case
      1. 1 + 1 = 2
   2. Test Overflow
      1. 0xFFFF + 0x1 = 0
3. AND
   1. Test Cases
      1. 0xFFFF & 0x0000 = 0x0000
      2. 0x1111 & 0x8888 = 0x0000
      3. 0x3333 & 0x1111 = 0x1111
4. BRANCH EQUAL
   1. Should Jump
      1. 1 == 1 Goto 0
      2. 0 == 0 Goto 1
   2. Should not Jump
      1. 1 == 0 Goto 0
      2. 0xFFFF == 0x1111 Goto 0
5. BRANCH NOT EQUAL
   1. Should Not Jump
      1. 1 != 1 Goto 0
      2. 0 != 0 Goto 1
   2. Should Jump
      1. 1 != 0 Goto 0
      2. 0xFFFF != 0x1111 Goto 0
6. JUMP
   1. Check value is written into PC
      1. 0x0000
      2. 0xFFFF
      3. 0x1111
   2. Examples
      1. 0x0000
      2. 0xFFFF
7. JUMP AND LINK
   1. Check PC is in $ra after command executes
8. JUMP REGISTER
   1. Check that value in PC is same that is in the defined register
9. LOAD WORD
   1. Test good addresses
      1. Make sure returned data value is correct and put into the correct register
   2. Test Bad addresses
      1. Exception should be raised
10. OR
    1. Test Cases
       1. 0x0000 | 0xFFFF = 0xFFFF
       2. 0x9999 | 0x6666 = 0xFFFF
11. SET LESS THAN
    * 1. 1 < 1 = 0
      2. 1 < 2 = 1
      3. -1 < 0 = 1
      4. 4 < 1 = 0
12. SHIFT LEFT LOGICAL
    1. Test Cases
       1. 8 << 2 = 32
       2. 7 << 2 = 28
       3. -2 << 1 = -1
13. SHIFT RIGHT LOGICAL
    1. Test Cases
       1. 8 >> 2 = 2
       2. 0xFFFF >> 4 = 0x0FFF
14. SHIFT RIGHT ARITHMETIC
    1. Test Cases
       1. 8 >> 2 = 2
       2. 0xFFFF >> 4 = 0xFFFF
15. SUBTRACT
    1. Test Case
       1. 1 – 2 = -1
       2. 2 – 1 = 1
       3. 2 - -1 = 3
16. STORE WORD
    1. Good Addresses
       1. Check to see data is in memory and value from register
    2. Bad Addresses
       1. Raise Exception

**Milestone 3**



The PC is a 16-bit register which holds the Program Counter value. It takes Write input and a jump/branch input, where logic is done to decide whether or not to branch or jump to a certain line of execution in memory. The IR is a register which holds the 16 bit value of the instruction. It takes an input from memory and a 1 bit value IR write, this decides whether or not to write to the IR. It outputs a 16-bit value that is the Instruction. Next are the 1 bit muxes RD1 & RD2. They take 4-bit inputs, and based on a control input depending on the instruction, chooses which input to put into the Read2 and Read3 inputs of the Reg File. Speaking of the Reg File, it is a 16 register file that takes 3 4-bit inputs, Read1,2,3, a 16-bit input writeData, and a control input regWrite. The Read inputs feed into 3 respective 16-bit 16 choice muxes that output Data1,2,3. There also exists a decoder for when writing to a register chosen at Read1 (and regWrite is 1). The main ALU does ALU ops based on the ALU control input and the two values that are going to be inputted. It is comprised of many 1-bit ALU’s strung together to do different operations. It also has a 1-bit output zero which is true when the two outputs were subtracted and are equal to one another. It feeds into an and gate when deciding when to branch. The last large component, memory, is similar to what was done for lab 7. It has memRead and memWrite control inputs, a ?-bit addr input, a 16-bit write input, and a 16-bit R-out output. We do not know how big the memory file is going to be, but we only want to make it big enough so that it will fit all of our instructions and any values that we need to save, but not so big that we wait 3 years waiting for it to be synthesized/emulated.

Default Memory for testing:

Address : Value

0x0000 : 0xFFFF

0x0002 : 0x0000

0x0004 : 0x1111

0x0006 : 0x1234

Unit Testing

1. Program Counter Register (PC) | Instruction Register (IR) | ALUout Register
   1. Set input value to 0x0000 and turn on the write control.
   2. Check that output is 0x0000.
   3. Change input to 0xFFFF.
   4. Check that output is 0xFFFF.
   5. Turn off write control and change input to 0x0000.
   6. Check that output is still 0xFFFF
2. Memory
   1. Turn on MemRead and turn MemWrite to off. Set Addr to 0x0000.
   2. Check that Dataout is 0xFFFF
   3. Change Addr to 0x0002
   4. Check that Dataout is 0x0000
   5. Flip MemRead and MemWrite. Set Addr to 0x000. Set WriteData to 0x0000
   6. Flip MemRead and MemWrite. Check Dataout is 0x0000
3. Register File
   1. Set Regwrite to on. Set Instruction wire to 0x0123. Set WriteData to 0xFFFF
   2. Check that C Register is 0xFFFF
   3. Change Instruction wire to 0x0321. Set WriteData to 0x1111
   4. Check that A Register is 0xFFFF and C Register is 0x1111
   5. Change Instruction wire to 0x0330. Set WriteData to 0xFFFF
   6. Check that A,B Registers are 0xFFFF and C Register is 0x0000
   7. Change Instruction wire to 0x0333. Set WriteData to 0x0000. Set Regwrite to off.
   8. Check that A,B,C Registers are 0x FFFF.

ALU Control Input / Output Values

1. A, B, C, MDR Registers
   1. Set input to 0xFFFF
   2. Check output is 0xFFFF
   3. Set input to 0x0000
   4. Check output is 0x0000
2. ALU

Refer to the table to the side for ALUControl Out codes which are

the ALU op Codes

* 1. Add
     + 0x0001 + 0x0001 = 0x0002
     + 0x0001 + 0xFFFF = 0x0000
  2. Sub
     + 0x0001 – 0x0001 = 0x0000
     + 0x0001 – 0x0002 = 0xFFFF
  3. And
     + 0x0000 & 0xFFFF = 0x0000
     + 0xCCCC & 0xCCCC = 0xCCCC
     + 0x8888 & 0xFFFF = 0x8888
  4. Or
     + 0x0000 & 0xFFFF = 0xFFFF
     + 0x1111& 0xEEEE = 0xFFFF
  5. Slt
     + 0x0000 < 0x0001 = 0x0001
     + 0x0000 < 0xFFFF = 0x0000
  6. Sll
     + 0x0001 << 1 = 0x0002
     + 0xFFFF << 4 = 0xFFF0
  7. Srl
     + 0xFFFF >> 4 = 0x0FFF
     + 0x0001 >> 1 = 0x0000
  8. Sra
     + 0xFFFF >> 4 = 0xFFFF
     + 0x0001 >> 1 = 0x0000

1. ALU Control
   1. For each Instruction[15:12] check to see the output code matches the table.
2. MemtoReg | IoR | RegDst1 | RegDst2 | ALUSrcA (1 bit Mux)
   1. Set Control to 0. Set Wire 0 to 0x0000, Wire 1 to 0xFFFF.
   2. Check Output is 0x0000
   3. Set Control to 1.
   4. Check Output is 0xFFFF
3. ALUSrcB Mux
   1. Set Control to 0. Set Wire 0 to 0x0000, Wire 1 to 0xFFFF, Wire 2 to 0x0002, Wire 3 to 0x1111.
   2. Check Output is 0x0000
   3. Set Control to 1.
   4. Check Output is 0xFFFF
   5. Set control to 2.
   6. Check Output is 0x0002
   7. Set control to 3.
   8. Check Output is 0x1111
4. PCSource Mux
   1. Set Control to 0. Set Wire 0 to 0x0000, Wire 1 to 0xFFFF, Wire 2 to 0x000.
   2. Check Output is 0x0000
   3. Set Control to 1.
   4. Check Output is 0xFFFF
   5. Set control to 2.
   6. Check Output is 0x0002

Datapath Testing

1. PC + ALU (Any Instruction)
   1. Test to see that PC is incremented by 2.
2. PC + IR + Memory (Any Instrution)
   1. See that Instruction that the value of PC is pulled into IR.
3. PC + IR + Memory + Reg File + ALU (0x0123)
   1. Test that a loaded command outputs the correct values to A(0x1), B(0x2), C(0x3) Registers and then the ALU adds them together and stores back in Read 3(0x3) Register.

Control Signals:

All Control signals are 1-bit unless otherwise specified

1. Regwrite:

This control signal turns on writing for the registers, A, B, C. 1 is write, 0 is not write.

1. Memread:

This control signal turns on direct memory reading. 1 is read, 0 is don’t read.

1. Memwrite:  
   This control signal turns on direct memory writing. 1 is write, 0 is don’t write.
2. ASA:  
   This control signal determines the source for the first ALU input. 0 is PC, 1 is register A
3. ASB:  
   This 2-bit control signal determines the source for the second ALU input. 00 is register B, 01 is register C, 10 is the immediate value, 2, and 11 is sign-extended address.
4. MTD:  
   this control signal determines the input for the memory file
5. PCwrite:  
   This control signal controls if PC is being written to. 1 is write, 0 is don’t write.
6. RD1:  
   This control signal controls the input for the second input into the register file. 0 is rs, 1 is rt
7. RD2:  
   This control signal controls the input for the third input into the register file. 0 is rd, 1 is rs
8. IOD:  
   This control signal changes the input for the address in memory access. 0 is from the PC, 1 is from the ALUout
9. PCsrc:  
   This 2-bit control signal determines what source the PC is being written from. 00 is the branch command’s source, 01 is from the ALU result, 10 is from ALUout
10. IRwrite:  
    This control signal turns on and off writing to the instruction register from memory. 1 is write, 0 is don’t write.
11. Branch:  
    This control signal determines if a branch is occurring. 1 is branch, 0 is don’t branch.
12. ALUoutC  
    This control signal controls the ALUout writing. 1 is write, 0 is don’t write.

ALU Op codes

|  |  |
| --- | --- |
| 000 | And |
| 001 | And |
| 010 | Or |
| 011 | Or |
| 100 | Add |
| 101 | Sub |
| 110 | Shift |
| 111 | Slt |