**Milestone 01**

**Design Document**

Due October 6, 2015

Adam Finer, Runzhi Yang, Katrina Kerrick, Fred Zhang

Contents

[Description of Registers Available 3](#_Toc432598981)

[English description of each machine language instruction format 4](#_Toc432598982)

[English Description of Each Instruction and Semantics 5](#_Toc432598983)

[Real Instructions: 5](#_Toc432598984)

[Pseudo Instructions: 7](#_Toc432598985)

[Explanation of Register Conventions 8](#_Toc432598986)

[Assembly Language Translation Rule 9](#_Toc432598987)

[Example Assembly Language programs 10](#_Toc432598988)

[Assembly Language Fragments for difficult instructions 10](#_Toc432598989)

[Euclid’s Algorithm 11](#_Toc432598990)

[Extra Example Assembly Language programs 12](#_Toc432598991)

[Descriptions of all of the Datapath Components: 15](#_Toc432598992)

[RTL Detail Table 18](#_Toc432598993)

[rtl Testing methods 20](#_Toc432598994)

# Description of Registers Available

There are sixteen registers total.

There are eight “t” registers, which are temporary variables. These registers can be overwritten at any time and do not need to be backed up on the stack. They are also used as parameters for functions. For example, if there is a function add(num1, num2, num3), num1 would be stored in $t0, num2 would be in $t1, and num3 would be in $t2.

There are four “s” registers. These registers must be backed up before use and restored after use. S registers are general purpose and can be used at any time.

There are two “v” registers. These registers function exactly like t registers, in that they don’t need to be backed up and can be used for general purposes. However, whenever a function is called, the v registers will contain the return value of the function.

There is one “at” register. This register is special use for when the compiler needs to split a pseudo-instruction into multiple instructions. This register may not be used in normal code.

There is one “ra” register. This register is for the return address of a function. When a function is called, the program counter is saved into the ra register. The program counter is then changed to the address of the function. Ra is a reserved register and may not be used for general purpose needs. Ra must be backed up on the stack before calling a function and restored after the call is complete.

The v0 register is also the display register. This is the final output number and will be displayed to the user.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| REGCODE | 00 | 01 | 10 | 11 |
| 00 | t0 | t1 | t2 | t3 |
| 01 | t4 | t5 | t6 | t7 |
| 10 | s0 | s1 | v0 | v1 |
| 11 | s2 | ra | ra | s3 |

There are two other registers not accessible through user interface.

We have PC register to keep track of which instruction we are executing. PC would increment by 2 to the next instruction for each instruction executed.

We have one separate “sp” register, but not accessible through assembly. This register is the stack pointer register, and keeps track of the growing end of the stack. It is only updated by the commands push and pop.

# English description of each machine language instruction format

All R, B, and J type instructions use a three-bit condition code. This three-bit code specifies one of the following:

|  |  |  |
| --- | --- | --- |
| **explanation** | **cc** | **condition** |
| greater than | 100 | > |
| greater than or equal | 110 | >= |
| less than | 001 | < |
| less than or equal | 011 | <= |
| equal | 010 | = |
| not equal | 101 | != |
| always execute | 111 |  |

How condition code work:

There is a three-bit flag register inside our CPU, which stores the last comparison result. The only cmd that would update flag register is cmp.

cmp $t0 $t1

In the above example, cmp would subtract $t0 and $t1 and set flag bits depending on the result. Three bits of flag represent negative(smaller than), zero(equal), positive(bigger than) respectively. Only one big of those flags can be 1 since $t0 - $t1 can only negative number, zero or positive number.

Whenever an instruction is executed (except I-type, without CC), we compare those condition bits and flag bits. Similarly, three CC bits represent negative(smaller than), zero(equal), positive(bigger than). As shown in the table above, if the flag bit indicate that the last comparison satisfies the condition, this instruction is going to be executed, otherwise, it is going to be skipped.

This cool feature enable us to do short ‘if’ statement without branches. We can do a compare and then skip instruction that we do not want to execute. Instruction skipped would take only two cycles, so a pretty efficient way to do single line ‘if’.

By default, the condition code is 111 if it is not specified in the instruction, so these instructions are always going to be executed.

R-type -- regular instruction format with two 4-bit register (r1 and r2), 1-bit memory-load-option-code (LM), and a 3-bit cc. Most of the time, instructions of this format will perform general operations with the values in r1 and r2, like add and or, and stores the result into r1. If one wants to use the CC, refer to the previous explanation. If LM is 1, then it would perform the operation with Mem[R[r2]] instead of R[r2], so just go to memory to load value.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| op(4) | r1(4) | r2(4) | LM(1) | CC(3) |

I-type -- immediate instruction format with one 4-bit register (r1) and 8-bit immediate value (imm). It performs a general operation of values in r1 and imm and stores it into r1.

|  |  |  |
| --- | --- | --- |
| op(4) | r1(4) | imm(8) |

B-type -- branch instruction format with a 3-bit condition code (CC) and a branch address (b-addr). This instruction compares flag and CC to find out whether it needs to branch or not and branch if necessary. Please refer to the previous explanation on cc.

|  |  |  |
| --- | --- | --- |
| op(4) | b-addr(9) | cc(3) |

J-type -- jump instruction format contains the operation code(op), the register to jump to(jr), a j-op code (jop), and the condition code (cc). Please refer to the previous explanation on cc.

|  |  |  |  |
| --- | --- | --- | --- |
| op(4) | jr(4) | jop(5) | cc(3) |

The jop refers to certain jump instructions by a 5 digit binary identification number. Here are the ids and their corresponding instructions:

|  |  |
| --- | --- |
| instruction | jop |
| jr | 00000 |
| jalr | 00001 |

All of these instructions have the same op code.

# English Description of Each Instruction and Semantics

## Real Instructions:

add -- adds a register and either a value from another register or a value from memory together and stores the value in the first register. This is an R-type instruction. All R-type can have optional condition code at its end.

add $r1, $r2 add $r1, ($r2) add $r1, $r2, 110

addi -- adds a register to an immediate value and stores the result in the register. This is an I-type instruction.

addi $r1, imm (immediate is 8 bits)

sto -- takes the value in a register and then stores it in memory at a specified address. That address is either from a register or a location in memory. This is an R-type instruction.

sto $r1, $r2 sto $r1, ($r2) sto $r1, $r2, 100

lui -- takes the immediate value and places it in the upper portion of whatever register specified. This is an I-type instruction.

lui $r1, 8-bit immediate

sub -- takes a register and subtracts either a value from another register or memory from it and stores the value in the register. This is an R-type instruction.

sub $r1, $r2 sub $r1, ($r2) sub $r1, $r2, 100

cmp -- takes a register and another register or a value from memory and compares their values. It then sets the three flag bits to show whether the first register is smaller (100), equal (010), or greater (001) than the second. This is an R-type instruction, so it also takes CC. We can optional skip cmp, so we can do multiple comparison in a row.

cmp $r1, $r2 cmp $r1, ($r2) cmp $r1, $r2, 100

cp -- takes one register and copies its value into a different register in the format of destination register, source register. This is an R-type instruction.

cp $r1, $r2 cp $r1, ($r2) cp $r1, $r2, 100

cpi -- initializes a register with an immediate value. This is an I-type instruction.

cpi $r1, imm (immediate is 8 bits)

and -- ands a register and either a value from another register or memory together and stores the value in the first register. This is an R-type instruction.

and $r1, $r2 and $r1, ($r2) and $r1, $r2, 100

xor -- takes a register value and another register or a value from memory, xors them together, and stores the new value in the first register. This is an R-type instruction.

xor $r1, $r2 xor $r1, ($r2) xor $r1, $r2, 100

push -- decrements the stack pointer register and then stores the value in the register in the stack. This is an R-type instruction.

push $r1 push $r1, 100

pop -- loads the value from the stack at the location of the stack pointer into the register and then increments the stack pointer. This is an R-type instruction.

pop $r1 pop $r1, 100

or -- Ors a register and either a value from another register or memory together and stores the value in the first register. This is an R-type instruction.

or $r1, $r2 or $r1, ($r2) or $r1, $r2, 100

jalr -- sets $ra to be PC, then jumps to a register. It is used to call functions that are at the location in instruction memory that is contained within the register argument. This is a J-type instruction.

jalr $r1 jalr $r1, 001

jr -- takes the value in a register and jumps to that instruction number. This is a J-type instruction.

jr $r1 jr $ra, 010

b -- takes one label and CC. As usual, CC determines whether to branch or not. If we leave out CC, it is an unconditional branch, operates like j in MIPS. This is a B-type instruction.

b label, 100

## Pseudo Instructions:

andi -- ands a value from a register and an eight bit immediate and puts the value into the source register.

andi $r1, imm (8 bit immediate)

cpi $at, imm (8 bit immediate)

and $r1, $at

ori -- ors a value from a register and an eight bit immediate and puts the value into the source register.

ori $r1, imm (8 bit immediate)

cpi $at, imm (8 bit immediate)

or $r1, $at

xori -- xors a value from a register with an eight bit immediate and puts the value in the source register.

xori $r1, imm (8 bit immediate)

cpi $at, imm (8 bit immediate)

xor $r1, $at

clear -- sets a register to have a value of zero.

clear $r1

cpi $r1, 0

addi (big) -- adds a 16 bit immediate to a value in a register.

addi $r1, imm (16 bit immediate)

lui $at, upper immediate

addi $at, lower immediate

add $r1, $at

cpi (big) -- sets a register to a 16 bit immediate value.

cpi $r1, imm (16 bit immediate)

lui $r1, upper immediate

addi $r1, lower immediate

cmpi -- compares a value from a register with an immediate value.

cmpi $r1, imm (16 bit immediate)

cpi $at, immediate

cmp $r1, $at

cmpi (big) -- compares a value from a register with an immediate value.

cmpi $r1, imm (8 bit immediate)

lui $at, upper immediate

addi $at, lower immediate

cmp $r1, $at

call -- jumps to a label and saves the return address. Used for calling functions.

call label

lui $ra, upper label

addi $ra, lower label

jalr $ra

j (big) -- jumps to a label unconditionally. For small distance, simply use b label

j label

lui $at, upper label

cpi $at, lower label

jr $at

beq -- branches to a label if two registers are equal.

beq $r1, $r2, label

cmp $r1, $r2

b label, 010

bne -- branches to a label if two registers are not equal.

bne $r1, $r2, label

cmp $r1, $r2

b label, 101

blt -- branches if the first register is less than the second register.

blt $r1, $r2, label

cmp $r1, $r2

b label, 100

bgt -- branches if the first register is greater than the second register

bgt $r1, $r2, label

cmp $r1, $r2

b label, 001

ble -- branches if the first register is less than or equal to the second register

ble $r1, $r2, label

cmp $r1, $r2

b label, 110

bge -- branches if the first register is greater than or equal to the second register

bge $r1, $r2, label

cmp $r1, $r2

b label, 011

# Explanation of Register Conventions

In general purpose assembly, the t and v registers may be used. However, these values are potentially destroyed whenever a function call is made, so their values must be saved on the stack or into an s register. The t registers are also used as parameters for function calls, starting with the lowest t register first. If more than eight parameters are needed, they are to be stored on the stack. The v registers are used as output from functions.

The s registers may be used in general purpose assembly code as well, but they must be backed up before use and restored after use. In exchange, their value is guaranteed across function calls.

The at register is for the compiler’s use only. When psuedo-instructions are changed into a set of regular instructions, if a temporary variable is needed, at is to be used. It is very important that at is not used for general purpose programming.

The ra register is for the return address of a function only. If it is ever changed, it must be backed up first and restored afterward. This allows for the nesting of function calls. For example, if main calls add(), and add() calls sub(), add() must back up the ra value from main and restore it after the sub() call has finished. Additionally, any t or v registers that hold important values must be backed up or their values will be lost.

The sp register is for the stack pointer. It is not accessible as general purpose register, and only modifiable via push and pop. It must be decremented when an item is pushed to the stack and incremented again when an item is restored. It is imperative that the stack pointer be the same at the beginning and end of each function!

# Assembly Language Translation Rule

For each instruction in the Assembly language, first translate the name of the instruction to an opcode using this table. All instruction format except I-type would have 3 bits conditional code (CC) at the end. If not specified with the assembly, those three bits is going to be 111 defaultly,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OPCODE | 00 | 01 | 10 | 11 |
| 00 | add [R] | addi [I] | sto [R] | lui [I] |
| 01 | sub [R] | cmp [R] | cp [R] | cpi [I] |
| 10 | and [R] | xor [R] | push [R] | pop [R] |
| 11 | or [R] | (blank) | jr/jalr [ J ] | b [B] |

After the first four bits are determined, the rest of the instruction is determined based on what type of instruction it is.

If it’s an R type:

Translate the two register names into register codes and append them. If the second

register had parenthesis around it symbolizing a retrieve from memory, then the next bit (LM) is one. Otherwise, it is zero. The last three bits are condition code (CC) used to decide whether to skip this R-type instruction or not.

opcode (4) + r1 (4) + r2 (4) + LM (1) + CC(3)

If it’s a B type:

The next 9 bits are for branch address and the last three bits are condition bits (CC) used to determine whether to branch or not. BranchAddress will be shift left one bit first and then add to PC to form the new PC address to jump to.

opcode (4) + branchAddress (9) + CC(3)

If it’s an I Type:

The next four bits are the register address. After that is an eight bit immediate. I-Type is the only type without CC.

opcode (4) + r1 (4) + imm(8)

If it’s a J Type:

The next four bits are the register address, with destination PC value. Then we have 5-bit jop. If it is jr, then last bit of jop is 0, or if it is jalr, then last bit of jop is 1. The last three bits are CC.

opcode (4) + jr(4) + jop(5) + CC(3)

# Example Assembly Language programs

## Assembly Language Fragments for difficult instructions

|  |  |
| --- | --- |
| \_clear $t0  sub $t0, $t0 | 0100000000000111 |
| \_beq $t1, $t2, label  cp $at, $t1  sub $at, $t2  b label, 101 | 0110110100010111  0100110100100111  1111(label)101 |
| \_blt $t1, $t2, label  cp $at, $t1  sub $at, $t2  b label, 100 | 0110110100010111  0100110100100111  1111(label)100 |
| \_ble $t1, $t2, label  cp $at, $t1  sub $at, $t2  b label, 110 | 0110110100010111  0100110100100111  1111(label)110 |
| \_addi(big) $t1, big  lui $at, upperbig  cpi $at, lowerbig  add $t1, $at | 0011110100000000  0111110100000000  0000000111010000 |
| \_li(big) $t1, big  lui $t1, upperbig  cpi $t1, lowerbig | 0011000100000000  0111000100000000 |
| \_cp(big) $t1, (big)  lui $t1, upperbig  cpi $t1, lowerbig | 0011000100000000  0111000100000000 |

## Euclid’s Algorithm

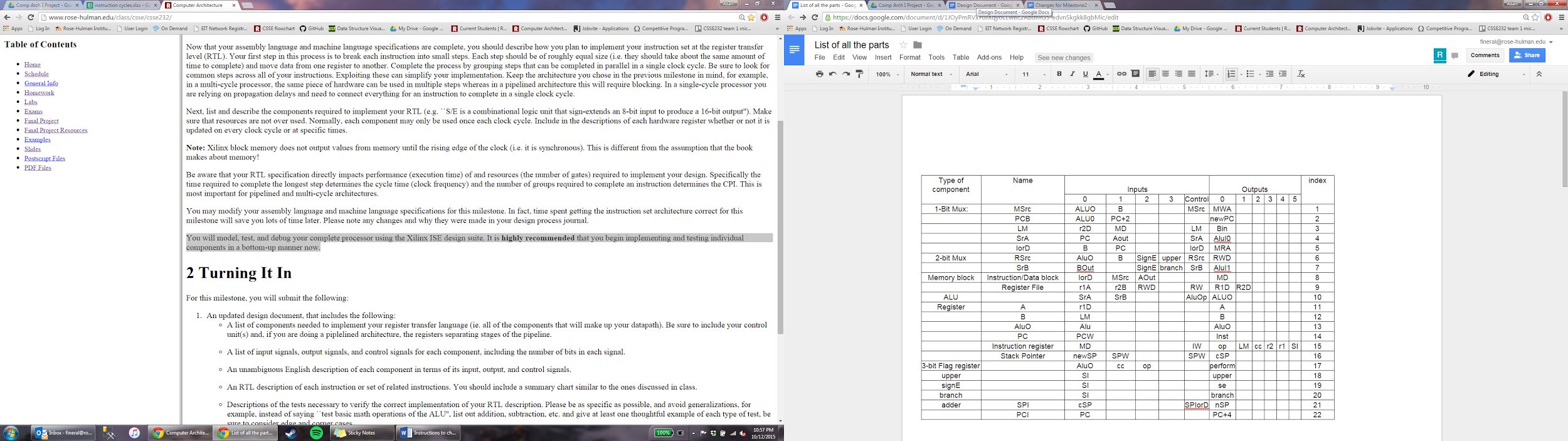
|  |  |
| --- | --- |
| **Assembly code** | **Binary Representation** |
| \_\_\_relPrime: | \_\_relPrime: |
| push $ra | 1010111000000111 |
| push $s0 | 1010100000000111 |
| push $s1 | 1010100100000111 |
| cp $s0, $t0 | 0110100000000111 |
| cpi $s1, 2 | 0111100100000010 |
| WHILE1: | WHILE1: |
| cp $t0, $s0 | 0110000010000111 |
| cp $t1, $s1 | 0110000110010111 |
| lui  $ra, \_\_gcd[15:8] | 0011111000000000 |
| addi $ra, \_\_gcd[7:0] | 0001111001100000 |
| jr $ra, jalr | 1110111000001111 |
| addi $s1, 1 | 0001100100000001 |
| cpi  $at, 1 | 0111110100000001 |
| cmp  $v0, $at | 0101101011010111 |
| b WHILE1, 101 | 1011100000000111 |
| cp $v0, $s1 | 0110101010010111 |
| pop $s1 | 1011100100000111 |
| pop $s0 | 1011100000000111 |
| pop $ra | 1011111000000111 |
| jr $ra, jr | 1110111000000111 |
| \_\_gcd: | \_\_gcd: |
| cpi   $t7, 0 | 0111011100000000 |
| cmp  $t0, $t7 | 0101000001110111 |
| cp   $v0, $t1, 010 | 0110101000010010 |
| jr   $ra, jr, 010 | 1110111000000010 |
| cmp  $t1, $t7 | 0101000101110111 |
| b RET, 010 | 1111011100011010 |
| WHILE2: | WHILE2: |
| cmp  $t0, $t1 | 0101000000010111 |
| sub  $t0, $t1, 001 | 0100000000010001 |
| sub  $t1, $t0, 110 | 0100000100000110 |
| cmp  $t1, $t7 | 0101000101110111 |
| b    WHILE2, 101 | 1111001110000101 |
| RET: | RET: |
| cp $v0, $t0 | 0110101000000111 |
| jr $ra, jr | 1110111000000111 |

## Extra Example Assembly Language programs

Linear sort

|  |  |  |
| --- | --- | --- |
| **Address** | **Assembly** | **comments** |
| 0x0000 | .globl \_\_main |  |
| 0x0004 | .globl A |  |
| 0x0008 | .globl N |  |
| 0x000c | .data |  |
| 0x0010 | A: .word 20, 56, -90, 37, -2, 30, 10, -66, -4, 18 |  |
| 0x0014 | N: .word 10 |  |
|  |  |  |
| 0x0018 | \_\_main: |  |
| 0x001c | addi $sp, -4 |  |
| 0x0020 | sto $ra, $sp |  |
| 0x0024 | la $s0, N |  |
| 0x0028 | la $s1, A |  |
| 0x002c | loop1: |  |
| 0x0030 | addi $sp, -4 | #move the stack and store N |
| 0x0034 | sto $s0, $sp |  |
| 0x0038 | cp $t1, $s0 | # put the variables in $t registers, and treat them like $a |
| 0x003c | cp $t0, $s1 | # this is the base address of the array |
| 0x0040 | jal SortMaxBeforeN | # call the sorting algorithm |
| 0x0044 | cp $s1. ($sp) | #restore the stack |
| 0x0048 | addi $sp, 4 |  |
| 0x004c | addi $s1, -1 | #Decrement N |
| 0x0050 | sub $t3, $t3 | # create a zero register |
| 0x0054 | bne $s1, $t3, loop |  |
| 0x0058 | exitMain: |  |
| 0x005c | cp $ra, $sp |  |
| 0x0060 | addi $sp, 4 |  |
| 0x0064 | jr $ra |  |
|  |  |  |
|  |  | #t0 is the base address of the array |
|  |  | #t1 is the max index of the array |
| 0x0068 | \_\_SortMaxBeforeN: |  |
| 0x006c | addi $sp, -4 |  |
| 0x0070 | sto $ra, $sp | #store the return address |
| 0x0074 | addi $sp, -4 |  |
| 0x0078 | sto $s0, $sp | #now $s0-2 are available to user |
| 0x007c | addi $sp, -4 |  |
| 0x0080 | sto $s1, $sp |  |
| 0x0084 | addi $sp, -4 |  |
| 0x0088 | sto $s2, $sp |  |
| 0x008c | sub $s0, $s0 | #create iteration i starts at 0 t2 |
| 0x0090 | sub $s1, $s1 |  |
| 0x0094 | addi $s1, 1 | #create 1 in the register t5 in lab3 |
| 0x0098 | cp $t6, ($t0) | #grab the 1st elem of array’s max elem |
| 0x009c | sub $t6, $t6 | #$t7 now is the zero register |
| 0x00a0 | loop1: |  |
| 0x00a4 | beq $s0, $t2, exit | #check if i is less than N |
| 0x00a8 | addi $sp, -4 | #store the base address of array |
| 0x00ac | sto $t0, $sp |  |
| 0x00b0 | addi $sp, -4 | #store the max index of the array |
| 0x00b4 | sto $t1, $sp |  |
| 0x00b8 | addi $sp, -4 | #store the current max element into stack |
| 0x00bc | sto $t6, $sp |  |
| 0x00c0 | cp $t0, $s0 | #The number that is needed to be shifted |
| 0x00c4 | cpi $t1, 2 | #store the index as well as the number needs to be shifted |
| 0x00c8 | jal \_\_shiftLeft |  |
| 0x00cc | sto $s1, $v0 | #after shift |
| 0x00d0 | cp $t6, ($sp) | #restore the current max element |
| 0x00d4 | addi $sp, 4 |  |
| 0x00d8 | cp $t1, ($sp) | #restore the max index of the array |
| 0x00dc | addi $sp, 4 |  |
| 0x00e0 | cp $t0, ($sp) | #restore the base address of the array |
| 0x00e4 | addi $sp, 4 |  |
| 0x00e8 | add $s1, $t0 | # set $s1 to address of A[i] |
| 0x00ec | cp $s1, ($s1) | #set $s1 to A[i] |
| 0x00f0 | bbe $s1, $t6, ok |  |
| 0x00f4 | cp $t6, $s1 | #set max to A[i] |
| 0x00f8 | cp $t7, $s0 | #set max index to i |
| 0x00fc | ok: |  |
| 0x0100 | addi $s0, 1 | #increment i |
| 0x0104 | j loop1 |  |
| 0x0108 | exit: |  |
| 0x010c | add $t3, $t1 | #puts the index of the last element in $t3 |
| 0x0110 | addi $t3, -1 |  |
| 0x0114 | addi $sp, -4 | #this stores the max element in last index |
| 0x0118 | sto $t6, $sp |  |
| 0x011c | addi $sp, -4 |  |
| 0x0120 | sto $t0, $sp | #this saves the base address of array |
| 0x0124 | addi $sp, -4 | #this stores the return address |
| 0x0128 | sto $ra, $sp |  |
| 0x012c | addi $sp, -4 | #store the max index |
| 0x0130 | sto $t7, $sp |  |
| 0x0134 | cp $t0, $t3 | #shift $t3 by 4 |
| 0x0138 | cpi $t1, 2 |  |
| 0x013c | jal \_\_shiftLeft |  |
| 0x0140 | cp $t0, ($sp) | #restore the max index |
| 0x0144 | addi $sp, 4 |  |
| 0x0148 | addi $sp, -4 |  |
| 0x014c | sto $v0, $sp | #store $t3 in stack |
| 0x0150 | cpi $t1, 2 |  |
| 0x0154 | jal \_\_shiftLeft |  |
| 0x0158 | cp $t7, $v0 |  |
| 0x015c | cp $t3, ($sp) | #this restores $t3 |
| 0x0160 | addi $sp, 4 |  |
| 0x0164 | cp $ra, ($sp) | #this restores the return address |
| 0x0168 | addi $sp, 4 |  |
| 0x016c | cp $t0, ($sp) | #restore $t0 |
| 0x0170 | addi $sp, 4 |  |
| 0x0174 | add $t3, $t0 |  |
| 0x0178 | cp $t4, ($t3) |  |
| 0x017c | sto $t4, $t7 |  |
| 0x0180 | cp $t6, ($sp) | #this restore the max element in last index |
| 0x0184 | addi $sp, 4 |  |
| 0x0188 | sto $t6, $t3 |  |
| 0x018c | cp $s2,($sp) |  |
| 0x0190 | addi $sp, 4 |  |
| 0x0194 | cp $s1,($sp) |  |
| 0x0198 | addi $sp, 4 |  |
| 0x019c | cp $s0,($sp) |  |
| 0x01a0 | addi $sp, 4 |  |
| 0x01a4 | cp $ra, ($sp) |  |
| 0x01a8 | addi $sp, 4 |  |
| 0x01ac | jr $ra |  |
|  |  |  |
|  |  | #$t0 is the number that needs to be shifted |
|  |  | #$t1 is the bits the register needs to be shifted |
| 0x01b0 | \_\_shiftLeft: |  |
| 0x01b4 | sto $t1, $sp |  |
| 0x01b8 | sub $t3, $t3 | #create a zero register |
| 0x01bc | shiftLoop: |  |
| 0x01c0 | beq $t1, $t3, exitShift |  |
| 0x01c4 | add $t0, $t0 |  |
| 0x01c8 | addi $t1, -1 |  |
| 0x01cc | j shiftLoop |  |
| 0x01d0 | exitShift: |  |
| 0x01d4 | sto $v0, $t0 |  |
| 0x01d8 | jr $ra |  |

# Descriptions of all of the Datapath Components:

**￼￼**

1. This 1-bit mux, named MSrc, takes three inputs: spt, the output from the stack pointer; B, the output from the register file; and the MSrc bit from the control. This MUX outputs what is written to memory, if needed.
2. This 1-bit MUX, named PCB, takes three inputs: ALU0, which would be from a branch; PC+2, which is a normal PC increment; and the PCB bit from the control. This MUX outputs the new PC value for the next instruction.
3. This 1-bit MUX, named LM, takes three inputs: r2D, which is the data from the second register; mD, which is data taken from memory; and the LM bit from the control.This MUX outputs either register data or memory data depending on what is specified by the user.
4. This 1-bit MUX, named SrA, takes three inputs: r1D, which is data from the first register; PC, which is the value of PC; and the SrA bit from the control. This MUX outputs either the PC, if the user is manipulating the program counter, or r1D, if the user is using the value from the first register of the instruction.
5. This 1-bit MUX, named IorD (which stands for instruction or data), takes three inputs: PC, which is is the value of the program counter; MSrc, which is either the value from the stack pointer or the value from the B register; and the IorD bit from the control. This MUX outs either the PC, so that the Main Memory can retrieve that instruction; sp, so that main memory can get that information off of the stack; or B, which would be if an R-type instruction accessed memory to get the value.
6. This 2-bit MUX, named RSrc, takes five inputs: ALUO, the output from the ALU component; signE, which would be the sign extended 16 bit immediate; upper, a 16 bit value with 8 bit number in the first 8 significant bits followed by 8 0 bits; B, which is the value from the B register; and the RSrc bit from the control. The output is either ALUO if the user is just performing an R-type instruction; signE, if the user is doing either addi or cpi; upper, if the user is doing a lui instruction; or B, if the user is performing a copy or a pop.
7. This 2-bit MUX, named SrB, takes four inputs: Bin, the value from the B register; signE, a sign extended value; branch, the amount to increase pc by for a branch; and the SrcB bit from the control. The output is either Bin if f the user is performing an operation between two values, signE if the user is sign extending a value, or branch if the user is trying to perform a branch.
8. Instruction/Data block is the memory, this takes four inputs: MRA, which reads the specified address; MWA, which chooses a specified memory location to write to; MWD writes the specified data to the address specified in MWA; and the MW control bit. The output of this memory block is MD, memory data, which is the data from the memory address specified in MRA.
9. Register File is a register block which takes four inputs: r1A, which is the address of $rs; r2B, which is the address of $rt; RWD, which is the value to write to the address specified in r1A; and the RW control bit. the output of this register is r1D, which is the data at r1A, and r2D, which is the data value at r2B.
10. Alu reads inputs from SrA and SrB, then computes the result according to ALUOp (the Alu operation code). The output of the ALU is ALUO which is the product of the operation done between SrA and SrB.
11. A is a register that stores the value of $rs. The input and output are both the value of $rs.
12. B is a register that stores the value of $rt. The input and output are both the value of $rt.
13. AluO is a register that stores the result of alu for future use. The input and output are both the same.
14. PC is a register that keeps track the location of the current instruction. The input and output are both the same
15. The instruction register reads the data coming out of the memory, and spits out a single instruction. The cc is used in R, J, and B type instructions. They are specified in the assembly code. r1 (4 bits) and r2 (4 bits) are both outputted as the register’s addresses specified in the instruction. The output for op (4 bits) code will simply send the information to the controls. The output for the LM (1 bit), load memory, only used in R-type instructions, is a 0 or 1 depending on whether or not the user accesses memory. SI is a 12 bit immediate, that could be used in load upper immediate, sign extension and branch.
16. Stack pointer: It takes the control bit SPW (stack pointer write) so the user can push and pop. It also take the input of an updated stack pointer that was either incremented or decremented by 2. The output of this register is the updated stack pointer received through the non-control input.
17. 3-Bit flag register: This flag register takes the input from Alu output, Control code, and operation code. The flag determines whether the next instruction will be executed or not. The operation code tells the instruction if it contains condition code, the condition code will determine whether the result match with the desire condition (see condition code specification). ALUO is used when condition code is used in instruction like add, it inputs zero, negative or positive, the and with the condition code. Next line will be executed if the anded result is not 0.
18. Upper: This circuit loads the upper 8 bit value of a register and outputs a 16 bit value with those 8bits in its most significant bits with 0s trailing.
19. signE: This circuit sign extends the value, so it become a 16 bits value.
20. Branch: This would shift the branch address right by 4 and sign extends it to a 16 bit, then feed to the SrcB mux.
21. SPIorD: This moves the stack pointer, it could increase by 2 so user can push item into the stack, or decrease by 2 to restore element from the stack.
22. PCI: This increment PC by 2, which prepares the cpu for the next instruction.

# RTL Detail Table

We use multi-cycle processor. For most processor there are around 4 cycles per instruction. Many immediate instruction only takes 2 cycle, such as lui, cpi to speed up those sudo instructions depending on them. Besides our rtl, we also include the control signals in the table. For those signals not specified, it has to be 0 if it control a register write, otherwise we don’t care about its value.

For all instruction except I-type, they will have Flag register compare CC and flag bits and output perform bit into control. When perform is 1, it means that flag bits satisfies CC and this instruction is going to be completed, otherwise, the control is going to transit to next instruction and skip this one.

For most R-type instruction except push and pop, there can be an extra cycle between Cycle 2 and Cycle 3 if LM = 1. In such case, they are going to pause for one cycle and wait register B to grab value from memory (B = Mem[B] LM = 1), after this extra cycle, the control is going back to Cycle 3 as usual. So the maximum cycle number is actually 5.

We decide to implement a mealy state instead of moore machine, so the control signal output depend not only on the state, but also the input signals. With this method, one control state can provide different control signals and different instructions can perform different work at Cycle 2. In the following diagram, there is a Cycle 2 default rtl and control signal. However, depending on op code, control would produce some special additional control bits (overriding the default)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | opcode | | Cycle 1 | Cycle 2 Default | Cycle 2 special | Cycle 3 | Cycle 4 |
| addi | 0001 | 1 | IR = Mem[PC]  ALUO = PC + 2  IorD = 1  IW | A = R[r1]  B = R[r2]  LM = 0 |  | ALUO = A + Sign(imm)  SrcA = 0  SrcB = e  ALUOp = (add) | R[r1] = ALUO  RW  RSrc = 0 |
| lui | 0011 | 3 | R[r1] = UpperImm  RW  RSrc = 2 |  |  |
| cpi | 0111 | 7 | R[r1] = SignExtend RW  RSrc = 1 |  |  |
| sto | 0010 | 2 |  | Mem[B] = A  MW  MSrc = 1 |  |
| cp | 0110 | 6 |  | R[r1] = B  RW  RSrc = 3 |  |
| add | 0000 | 0 |  | ALUO= A + B  SrcA = 0  SrcB = 0  ALUOp = (add) | R[r1] = ALUO  RW  RSrc = 0 |
| sub | 0100 | 4 |  | ALUO= A - B  SrcA = 0  SrcB = 0  ALUOp = (sub) | R[r1] = ALUO  RW  RSrc = 0 |
| cmp | 0101 | 5 |  | ALUO = A - B  update flag  SrcA = 0  SrcB = 0  ALUOp = (sub)  FU |  |
| and | 1000 | 8 |  | ALUO= A & B  SrcA = 0  SrcB = 0  ALUOp = (and) | R[r1] = ALUO  RW  RSrc = 0 |
| xor | 1001 | 9 |  | ALUO= A ^ B  SrcA = 0  SrcB = 0  ALUOp = (xor) | R[r1] = ALUO  RW  RSrc = 0 |
| or | 1100 | 12 |  | ALUO= A ! B  SrcA = 0  SrcB = 0  ALUOp = (and) | R[r1] = ALUO  RW  RSrc = 0 |
| push | 1010 | 10 | $sp -= 2  SPW  SPIorD = 0 | Mem[sp} = A  MW  MSrc = 0 |  |
| pop | 1011 | 11 | B = Mem[bp]  LM = 1  IorD = 0  MSrc = 0 | R[r1] = B  $sp +=2  RW  RSrc = 3  SPW  SPIorD = 1 |  |
| jr | 1110 | 14 |  | ALUO = A  SrA = 0  ALUOp = (A) | PC = ALUO  PCW |
| jal | ALUO = PC  SrcA = 1  ALUOp = (A) | ALUO = A  R[r1] = AlUO  SrA = 0  ALUOp = (A)  RW  RSrc = 00 | PC = ALUO  PCW |
| b | 1111 | 15 | ALUO = PC + branch(imm)  SrcA = 1  SrcB = branch  ALUOp = (add) | PC = ALUO  PCW |  |

# rtl Testing methods

for all instructions, besides the jumps and branches, make sure pc was incremented by 2 in all cases.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction | Opcode | Testing method | | |
| addi | 0001 | addi $t0, 5  The expected value of $t0 after the execution should be 0000 0000 0000 1001 |  |  |
| lui | 0011 | lui $t0, 0xff  The expected value of $t0 after the execution should be 1111 1111 0000 0000 |  |  |
| cpi | 0111 | cpi $r1, 0xff  The expected value of $ri is 0000 0000 1111 1111 |  |  |
| sto | 0010 | sto $r1, $r2  If $r1 is 0xffff, then $r2 will be 0xffff after the execution | sto $r1, ($r2)  If $r1 is 0xffff, $r2 is 0x0001, then when pc reads the address of 0x0001, it should be 0xffff | sto $r1, $r2, 100  If the cc is true, then the $r1 will be stored in $r2, else it will skip this instruction |
| cp | 0110 | cp $r1, $r2  Check to make sure that the values in the registers are both equal. | cp $r1, ($r2)  check to make sure the value in $r1 is equal to the value at $r2 in memory | cp $r1, $r2, 100  if the cc is true, do the check, if cc is false, make sure $r1 is not changed |
| add | 0000 | add $r1, $r2  check to make sure the values were added correctly. | add $r1, ($r2)  check to make sure the values were added correctly | add $r1, $r2, 110  if the cc is false, make sure the add was not done, otherwise do previous. |
| sub | 0100 | sub $r1, $r2  check to make sure the value were subtracted correctly. | sub $r1, ($r2)  check to make sure the values were subtracted correctly | sub $r1, $r2, 100  if the cc is false, make sure the subtraction was not done, otherwise do previous. |
| cmp | 0101 | cmp $r1, $r2  check to make sure the value in the flag register equals the result of this comparison | cmp $r1, ($r2)  check to make sure the value in the flag register equals the result of this comparison | cmp $r1, $r2, 100  if the cc is false, make sure the value in the flag register was unchanged. otherwise do previous |
| and | 1000 | and $r1, $r2  make sure the values were anded correctly | and $r1, ($r2)  check to make sure the values were anded correctly | and $r1, $r2, 100  if the cc is false, make sure the value in $r1 is unchanged, otherwise do previous |
| xor | 1001 | xor $r1, $r2  If $r1 is 0x1, $r2 is 0x1, the result will be 0x0 and stored in $r1 | xor $r1, ($r2)  check $r1 if it is the address of $r2 xor with $r1’s original value | xor $r1, $r2, 100  is cc is false, make sure $r1 is unchanged |
| or | 1100 | or $r1, $r2  If $r1 is 0x11, $r2 is 0x11, then the $r1 will be 0x11 after the execution | or $r1, ($r2)  Check to make sure the value were or correctly | or $r1, $r2, 100  if cc is false, the $r1 should remain the same |
| push | 1010 | push $r1  Check whether $sp is incremented.  Check if Mem[$sp] = $r1 |  | push $r1, 100  if the cc is false, make sure $sp is not incremented, otherwise do previous. |
| pop | 1011 | pop $r1  Check whether $sp is decremented.  Check if Mem[$sp - 2] = $r1 |  | pop $r1, 100  if the cc is false, make sure $sp did not increment, otherwise do previous |
| jr | 1110 | jr $r1  check the PC if it is the same value as $r1 |  | jr $ra, 010  if the cc is false, then PC should be PC+2 |
| jalr | jalr $r1  check the PC to make sure the jump was executed correctly |  | jalr $r1, 010  if the cc is false, then PC should be PC+2 |
| b | 1111 |  |  | b label, 100  if the cc is true, make sure the branch was executed. |