**Milestone 01**

**Design Document**

Due October 6, 2015

Adam Finer, Runzhi Yang, Katrina Kerrick, Fred Zhang

Contents

[*Description of Registers Available* 3](#_Toc431908406)

[English description of each instruction and semantics 3](#_Toc431908407)

[Explanation of Register Conventions 5](#_Toc431908408)

[English description of each machine language instruction format 6](#_Toc431908409)

[Assembly Language Translation Rule 6](#_Toc431908410)

[Example Assembly Language programs 7](#_Toc431908411)

[Assembly Language Fragments for difficult instructions 8](#_Toc431908412)

[Extra Example Assembly Language programs 10](#_Toc431908413)

[Ideas for exception handling 13](#_Toc431908414)

# 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 three “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 output 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.

There is one “sp” register. This register is the stack pointer register, and keeps track of the growing end of the stack. At the end of each function, including, main, the value in sp must be the same as it was in the beginning of the method. Sp cannot be used for general purpose programming.

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

# English description of each instruction and semantics

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.

add $r1, $r2 or do add $r1, ($r2)

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 or do sto $r1, ($r2)

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 or do sub $r1, ($r2)

subi -- takes a register and subtracts an immediate value from it and stores the value in the register. This is and I-type instruction.

subi $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 or do and $r1, ($r2)

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

andi $r1, imm (immediate is 8 bits)

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 do or $r1, ($r2)

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

ori $r1, imm (immediate is 8 bits)

cp -- copy a register to another register. We need this operation, since we are doing something similar to accumulator and we need something to initialize our registers. This is an R-type instruction. By using parentheses, cp can somehow load data from memory.

cp $r1, $r2 or do cp $r1, ($r2)

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

cpi $r1, imm (immediate is 8 bits)

b -- takes the condition code, a 4 bit number (we are only use the first 3 bits now, the last bit might be used for detecting overflow later, but we only encode the first 3 in assembly language), then an instruction to jump to. Depending on the result of the previous line of code and what the condition code is determines whether we branch to the destination or not. Here is a quick chart to represent what condition code specifies.

pseudoInst CC condition

bbl 100 >

bbe 110 >=

blt 001 <

ble 011 <=

beq 010 =

bne 101 <>

This is a B-type instruction.

b 1100, label

jal -- jumEp to a specified label and save the current instruction address plus 4 in the return address register. This is a J-type instruction.

jal label

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

jr $r1

eret - return from exception handling and jump back to where the exception happened. It does not take any argument. It can only be used properly when interrupt level bit is 1.

eret

j -- takes a label, which corresponds to a certain line of code, and jumps to that instruction. This is a j type instruction.

j label

# 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 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!

# English description of each machine language instruction format

R-type -- regular instruction format with two 4-bit register (r1 and r2), 1-bit memory-load-option-code (LM). Most of the time, it performs general operation of values in r1 and r2, like add and or, and stores the result into r1. Flag would be set depending on the final result of the calculation. There are three unused bits in R-type. We may explore those bits as function code and add specification later.

There are a few exceptions for R-type: jr and eret. These two instruction do not do general purpose jumps but set PC instead. Flag bits are unchanged and last three bits are not always 0. For jr, it would be 000, but for eret, it would be eret. (We may make further changes to this part in the future)

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. Flag would be set depending on the final result of the calculation.

B-type -- branch instruction format with a 4-bit condition code (CC) and a branch address (b-address). This instruction compares flag and CC to find out whether it needs to branch or not and branch if necessary.

J-type -- jump instruction format with a 12-bit giant block of jump address. It basically jumps to that address in the instruction.

# Assembly Language Translation Rule

For each instruction in the Assembly language, first translate the name of the instruction to an opcode using this table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OPCODE | 00 | 01 | 10 | 11 |
| 00 | add [R] | addi [I] | sto [R] | lui [I] |
| 01 | sub [R] | subi [I] | cp [R] | cpi [I] |
| 10 | and [R] | andi [I] | b [B] | jal [J] |
| 11 | or [R] | ori [I] | jr / eret *[R]* | j [J] |

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. For now, the last three bits are unused and are zero except for eret.

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

However, there are two exceptions for jr and eret. jr only uses the spot r1 and ignore r2 and LM. eret ignores both r1, r2, LM. jr and eret have the same op code -- 1110, but jr has 000 as last three bits, while eret has 100 there.

If it’s a B type:

The next four bits (CC) are taken directly from the Assembly instruction. The next bit is

unused for now and is zero. The last bits are the immediate value translated to binary.

opcode (4) + CC (4) + branchAddress (8)

If it’s an I Type:

The next four bits are the register code. After that is an eight bit immediate.

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

If it’s a J Type:

The next value is an immediate.

opcode (4) + jumpAddress (12)

# Example Assembly Language programs

Euclid’s Algorithm

|  |  |  |
| --- | --- | --- |
| **Instruction Address** | **Assembly Code** | **Comments** |
| 0x0000 | \_\_relPrime: |  |
| 0x0004 | addi $sp, -4 | #store ra on the stack |
| 0x0008 | sto $ra, $sp |  |
| 0x000c | addi $sp, -4 | #store old $s0 on the stack |
| 0x0010 | sto $s0, $sp |  |
| 0x0014 | addi $sp, -4 | #store old $s1 on the stack |
| 0x0018 | sto $s1, $sp |  |
| 0x001c | cpi $s1, 2 | #int m = 2 |
| 0x0020 | cp $s0, $t0 | #put argument to a s register |
| 0x0024 | WHILE1: |  |
| 0x0028 | cp $t0, $s0 | #put variables into input registers |
| 0x002c | cp $t1, $s1 |  |
| 0x0030 | jal \_\_gcd | #do the function call to gcd |
| 0x0034 | subi $v0, 1 | #check to see if the output is not 1 |
| 0x0038 | b 010, DONE |  |
| 0x003c | addi $s1, 1 | #increments m |
| 0x0040 | j WHILE1 | #loops back |
| 0x0044 | DONE: |  |
| 0x0048 | cp $v0, $s1 | #saves m in the return register |
| 0x004c | cp $s1, ($sp) | #loads previous s1 |
| 0x0050 | addi $sp, 4 |  |
| 0x0054 | cp $s0, ($sp) | #loads previous s0 |
| 0x0058 | addi $sp, 4 |  |
| 0x005c | cp $ra, ($sp) | #loads previous ra |
| 0x0060 | addi $sp, 4 | #gets sp to original value |
| 0x0064 | jr $ra | #returns |
| 0x0068 | \_\_gcd: |  |
| 0x006c | subi $t0, 0 | #checks to see if a is not zero |
| 0x0070 | b 101, WHILE2 | #no- just begin return, yes- loop |
| 0x0074 | cp $v0, $t1 | #get return value |
| 0x0078 | jr $ra | #return |
| 0x007c | WHILE2: |  |
| 0x0080 | subi $t1, 0 | #checks to see if b is not 0 |
| 0x0084 | b 010, SKIP2 |  |
| 0x0088 | cp $t2, $t0 | #stores a so we do comparison |
| 0x008c | sub $t2, $t1 | #checks to see if b <a |
| 0x0090 | b 011, ELSE |  |
| 0x0094 | sub $t0, $t1 | #a=a-b |
| 0x0098 | j WHILE2 | #loops |
| 0x009c | ELSE: |  |
| 0x00a0 | sub $t1, $t0 | #b=b-a |
| 0x00a4 | j WHILE2 | #loops |
| 0x00a8 | SKIP2: |  |
| 0x00ac | cp $v0, $t0 | #saves return in register |
| 0x00b0 | jr $ra | #returns to relPrime |

# Assembly Language Fragments for difficult instructions

|  |  |
| --- | --- |
| Assembly code  \_clear $t0  sub $t0, $t0  \_beq $t1, $t2, label  cp $at, $t1  sub $at, $t2  b 010, label  \_blt $t1, $t2, label  cp $at, $t1  sub $at, $t2  b 100, label  \_ble $t1, $t2, label  cp $at, $t1  sub $at, $t2  b 110, label  \_addi(big) $t1, big  lui $at, upperbig  ori $at, lowerbig  add $t1, $at  \_li(big) $t1, big  lui $t1, upperbig  ori $t1, lowerbig  \_cp(big) $t1, (big)  lui $at, upperbig  ori $at, lowerbig  cp $t1, ($at) | Machine code translation  (\_clear)  0100 0000 0000 0000  (\_beq)  0110 1101 0001 0000  0100 1101 0010 0000  1010 0100 (label)  (\_blt)  0110 1101 0001 0000  0100 1101 0010 0000  1010 1000 (label)  (\_ble)  0110 1101 0001 0000  0100 1101 0010 0000  1010 1100 (label)  (\_addi)  0011 1101 (upperbig)  1101 1101 (lowerbig)  0000 0001 1101 0000  (\_li)  0011 0001 (upperbig)  1101 0001 (lowerbig)  (\_cp)  0011 1101 (upperbig)  1101 1101 (lowerbig)  0110 0001 1111 1000 |

|  |  |
| --- | --- |
| **Assembly code** | **Binary Representation** |
| \_\_relPrime: | \_\_relPrime: |
| addi $sp, -4 | 0001 1111 1111 1110 |
| sto $ra, $sp | 0010 1110 1111 0000 |
| addi $sp, -4 | 0001 1111 1111 1110 |
| sto $s0, $sp | 0010 1000 1111 0000 |
| addi $sp, -4 | 0001 1111 1111 1110 |
| sto $s1, $sp | 0010 1001 1111 0000 |
| cpi $s1, 2 | 0111 1001 0000 0010 |
| cp $s0, $t0 | 0110 1000 0000 0000 |
| WHILE1: | WHILE1: |
| cp $t0, $s0 | 0110 0000 1000 0000 |
| cp $t1, $s1 | 0110 0001 1001 0000 |
| jal \_\_gcd | 1011 0000 0001 1010 |
| subi $v0, 1 | 0101 1010 0000 0001 |
| b 010, DONE | 1010 0100 0001 0010 |
| addi $s1, 1 | 0001 1001 0000 0001 |
| j WHILE1 | 1111 0000 0000 1001 |
| DONE: | DONE: |
| cp $v0, $s1 | 0110 1010 1001 0000 |
| cp $s1, ($sp) | 0110 1001 1111 1000 |
| addi $sp, 4 | 0001 1111 0000 0100 |
| cp $s0, ($sp) | 0110 1000 1111 1000 |
| addi $sp, 4 | 0001 1111 0000 0100 |
| cp $ra, ($sp) | 0110 1110 1111 1000 |
| addi $sp, 4 | 0001 1111 0000 0100 |
| jr $ra | 1110 1110 0000 0000 |
| \_\_gcd: | \_\_gcd: |
| subi $t0, 0 | 0101 0000 0000 0000 |
| b 101, WHILE2 | 1010 1010 0010 1111 |
| cp $v0, $t1 | 0110 1010 0001 0000 |
| jr $ra | 1110 1110 0000 0000 |
| WHILE2: | WHILE2: |
| subi $t1, 0 | 0101 0000 0000 0000 |
| b 010, SKIP2 | 1010 0100 0001 1010 |
| cp $t2, $t0 | 0110 0010 0000 0000 |
| sub $t2, $t1 | 0100 0010 0001 0000 |
| b 011, ELSE | 1010 0110 0011 0111 |
| sub $t0, $t1 | 0100 0000 0001 0000 |
| j WHILE2 | 1111 0000 0010 1111 |
| ELSE: | ELSE: |
| sub $t1, $t0 | 0100 0001 0000 0000 |
| j WHILE2 | 1111 0000 0010 1111 |
| SKIP2: | SKIP2: |
| cp $v0, $t0 | 0110 1010 0000 0000 |
| jr $ra | 1110 1110 0000 0000 |

# 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 |  |

# Ideas for exception handling

We have EPC and exception register (ER) to store both cause and status. However, no instruction can read or write to EPC and ER directly.

When exception happens, hardware would automatically update EPC and ER. Then immediately, it backs up $t0 to the stack, put ER into $t0 and jumps to exception handler.

In this way, ER serves as the argument of exception handler. ER would contain information like exception code, interrupt level, interrupt enable, etc. If the exception handler needs to use register other than $t0, it needs to back and restore them. After the exception handler finishes its job, it needs to put the updated ER into $t0 and call eret. eret updates ER, restore $t0 to its old value, and then jumps to EPC.

In this way, we do not need special instruction to read and write to ER. Out CPU would load ER into $t0 when exception occurs and updates in eret.