*Course No. ELEC 5200/6200*

*Computer Architecture and Design*

Project: Part-1

Date: February 17, 2017

**

*Prepared by*

*Md. Mahabubul Alam*

*Banner ID: 903937005*

*Graduate Student, Electrical Engineering*

*Auburn University*

*Benjamin Cyr*

*Banner ID: 903466318*

*Senior, Computer Engineering*

*Auburn University*

***Project Requirements:***

1. The ISA may contain no more than 16 unique instructions. However, you may have multiple formats for a given type of instruction, if necessary.
2. Of the 16 instructions, at least one instruction should make your processor **HALT.**
3. The ISA is to support 16-bit data words only. (No byte operands.)

a. All operands are to be 16-bit signed integers (2’s complement).

b. Each instruction must be encoded using one 16-bit word.

1. The ISA is to support linear addressing of 1K, 16-bit words memory. The memory is to be word-addressable only - **not byte-addressable**.
2. The ISA should contain appropriate numbers and types of user-programmable registers to support it. Since this is a small processor, the hardware does not necessarily need to support dedicated registers for stack pointer, frame pointer, etc.
3. The ISA must “support” the following C Programming Language constructs:

* Assignment operator: *variable = expression*;
  + Supported arithmetic operators in expressions must include: add (+) and subtract (-)
    - It is not necessary to support multiply (\*) and divide (/)
  + Supported logical operators in expressions must include: and (&) and or (|)
  + Data are limited to:
    - 16-bit two’s-complement integers (Example: int a;)
    - One-dimensional integer arrays (Example: int a[10];)
* Control flow structures: “if-else” structures, “while” loops, “for” loops
  + These should support the six standard relational operators:

==, !=, >, <=, <, >=

* Functions (call and return), with parameters able to be passed by value or by reference.

1. Provide the following information about your ISA:

* List and describe the roles of the user-programmable registers.
* List and describe the different instruction formats used.
* For each instruction in your instruction set, list the following:
* Assembly language for each form of the instruction - mnemonic and operands
* Machine language for each form of the instruction: instruction code format, op-code, and operand encoding
* *Justification for including each form of the instruction in your ISA*

1. For each C construct listed in item 6 above, provide an example showing how the construct would be “compiled”, i.e. implemented with your instruction set, by writing an example of the C construct and the corresponding assembly language implementation.

***Instruction Set Architecture:***

The requirements for this project are to design a small processor that can still perform the basic functions needed by C programs. But before the process can be designed, the Instruction Set Architecture must be designed. It must be clear what functions the processor can perform, and how those functions fulfill the requirements of the project.

The first design decision is the number and type of instructions. As part of the requirements, there can be a maximum of sixteen instructions. The opcode was chosen to be four bits, as the only instruction type that can encode extra functions into the end of the instruction are R-Type instructions. Since there were eight instructions chosen as necessary that were not R-Type (HALT, LW, SW, BNE, BEQ, JL, J, JR), there would have to be at least nine opcodes. One for R-Type, and eight for non-R-Type. And since that requires four bits, then each of the sixteen instructions were just given a unique opcode. The sixteen instructions that were chosen are shown in Table 1 below.

Table 1: Instruction Set

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Opcode | Type | Description |
| HALT | 0000 | S-Type | Halts the CPU |
| ADD | 0001 | R-Type | Add two registers |
| SUB | 0010 | R-Type | Subtract one register from another |
| AND | 0011 | R-Type | Bitwise AND two registers |
| OR | 0100 | R-Type | Bitwise OR two registers |
| SLT | 0101 | R-Type | Set to 1 if one register is less than another |
| LSI | 0110 | I-Type | Logical/Arithmetic shift left immediate |
| RSI | 0111 | I-Type | Arithmetic shift right immediate |
| ADDI | 1000 | I-Type | Add immediate operand |
| LW | 1001 | I-Type | Load word from memory |
| SW | 1010 | I-Type | Store word from memory |
| BNE | 1011 | I-Type | Branch if two registers are not equal |
| BEQ | 1100 | I-Type | Branch if two registers are equal |
| JL | 1101 | J-Type | Jump to location and set link register |
| J | 1110 | J-Type | Jump to location within instruction |
| JR | 1111 | S-Type | Jump to location in register |

The reasons and format of each instruction will be described below.

***Arithmetic “R-Type” Instructions:***

ADD: Used to add two registers together. A staple in any architecture, as adding is one of the most important tasks a processor must be able to do.

Assembly: ADD $rd, $rs, $rt Function: $rd = $rs + $rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Rs | Rt | Rd | Reserved |
| 0001 | XXX | XXX | XXX | XXX |
| 4 | 3 | 3 | 3 | 3 |

SUB: Used to subtract one from register from another. The complement of adding, it is necessary for many programs to subtract one positive number from another. The only way to accomplish this would be a subtract instruction.

Assembly: SUB $rd, $rs, $rt Function: $rd = $rs - $rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Rs | Rt | Rd | Reserved |
| 0010 | XXX | XXX | XXX | XXX |
| 4 | 3 | 3 | 3 | 3 |

AND: Used to bitwise AND two registers. This fulfills the project requirements to support the “&” C construct. It is useful for masking bits, clearing bits and interacting with peripheral devices.

Assembly: AND $rd, $rs, $rt Function: $rd = $rs & $rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Rs | Rt | Rd | Reserved |
| 0011 | XXX | XXX | XXX | XXX |
| 4 | 3 | 3 | 3 | 3 |

OR: Used to bitwise OR two registers. This fulfills the project requirements to support the “|” C construct. It is useful for setting bits and interacting with peripheral devices.

Assembly: OR $rd, $rs, $rt Function: $rd = $rs | $rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Rs | Rt | Rd | Reserved |
| 0100 | XXX | XXX | XXX | XXX |
| 4 | 3 | 3 | 3 | 3 |

SLT: Sets the destination register to “1” if the first source register is less than the second source register. Clears the destination register to “0” otherwise. This is the primary instruction for branch conditions. It is necessary for the four required condition checks: >, <, >=, <=.

Assembly: SLT $rd, $rs, $rt Function: $rd = $rs < $rt ? 1 : 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Rs | Rt | Rd | Reserved |
| 0101 | XXX | XXX | XXX | XXX |
| 4 | 3 | 3 | 3 | 3 |

***Immediate “I-Type” Instructions:***

LSI: Performs a left shift on the given register by the given amount. Useful in performing calculations involving powers of two, and helps in manipulating bit patterns.

Assembly: LSI $rt, $rs, [0 to 16] Function: $rt = $rs << [0 to 16]

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs | Rt | Immediate |
| 0110 | XXX | XXX | XXXXXX |
| 4 | 3 | 3 | 6 |

RSI: Performs a right shift on the given register by the given amount. Useful in performing calculations involving powers of two, and helps in manipulating bit patterns.

Assembly: RSI $rt, $rs, [0 to 16] Function: $rt = $rs >> [0 to 16]

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs | Rt | Immediate |
| 0111 | XXX | XXX | XXXXXX |
| 4 | 3 | 3 | 6 |

ADDI: Adds an immediate operand to the given register. Useful in simple calculations, as memory and registers are not required to store small constants.

Assembly: ADDI $rt, $rs, [-32 to 31] Function: $rt = $rs + [-32 to 31]

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs | Rt | Immediate |
| 1000 | XXX | XXX | XXXXXX |
| 4 | 3 | 3 | 6 |

LW: Load a word from memory into a register. A necessary operation, as registers cannot hold all the necessary data in most programs.

Assembly: LW $rt, [-32 to 31]($rs) Function: $rt = \*($rs + [-32 to 31])

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs | Rt | Immediate |
| 1001 | XXX | XXX | XXXXXX |
| 4 | 3 | 3 | 6 |

SW: Store a word from a register into memory. A necessary operation, as registers cannot hold all the necessary data in most programs.

Assembly: SW $rt, [-32 to 31]($rs) Function: \*($rs + [-32 to 31]) = $rt

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs | Rt | Immediate |
| 1010 | XXX | XXX | XXXXXX |
| 4 | 3 | 3 | 6 |

BNE: Add an offset to the PC counter if the two operands are not equal. Necessary to fulfill the branch condition !=, and used along with SLT for > and < conditions.

Assembly: BNE $rs, $rt, [-32 to 31] Function: $pc = ($rs != $rt) ? $pc + [-32 to 31] : $pc + 1

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs | Rt | Immediate |
| 1011 | XXX | XXX | XXXXXX |
| 4 | 3 | 3 | 6 |

BEQ: Add an offset to the PC counter if the two operands are equal. Necessary to fulfill the branch condition ==, and used along with SLT for >= and <= conditions.

Assembly: BEQ $rs, $rt, [-32 to 31] Function: $pc = ($rs == $rt) ? $pc + [-32 to 31] : $pc + 1

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | Rs | Rt | Immediate |
| 1100 | XXX | XXX | XXXXXX |
| 4 | 3 | 3 | 6 |

***Jumping “J-Type” Instructions:***

JL: Jumps to the location given in the opcode by loading the immediate into the bottom twelve bits of the program counter. It also stores the current value of $pc + 4 into the link register to enable a return from a function call.

Assembly: JL [0 to 4095] Function: $pc = concatenate( $pc[15:12], [0 to 4095] )

|  |  |
| --- | --- |
| Opcode | Immediate |
| 1101 | XXXXXXXXXXXX |
| 4 | 12 |

J: Jumps to the location given in the opcode by loading the immediate into the bottom twelve bits of the program counter.

Assembly: J [0 to 4095] Function: $pc = concatenate( $pc[15:12], [0 to 4096] )

|  |  |
| --- | --- |
| Opcode | Immediate |
| 1110 | XXXXXXXXXXXX |
| 4 | 12 |

***Special “S-Type” Instructions:***

JR: Jumps to the location in the given register. Necessary for returning from a function call, as you need to jump to the location in the link register.

Assembly: JR $rs Function: $pc = $rs

|  |  |  |
| --- | --- | --- |
| Opcode | Rs­ | Reserved |
| 1111 | XXX | XXXXXXXXX |
| 4 | 3 | 9 |

HALT: Stops the program.

Assembly: HALT Function: $pc = $pc

|  |  |
| --- | --- |
| Opcode | Reserved |
| 0000 | XXXXXXXXXXXX |
| 4 | 12 |

***Register Roles:***

The architecture was chosen to have only eight registers. This would give us a comfortable number of registers to work with for simple calculations, and only takes up three bits of addressing per register. This was necessary for I-Type instructions to give a range of immediate values of six bits, from

-32 to 31. If there were sixteen registers, the range of immediate values would only be four bits, from -8 to 7, which does not give much range for branching. Any branches larger than eight memory locations away would require a jump. But with eight registers, there is a wider range for branching.

The roles of the registers are described in Table 2 below.

Table 2: Roles of the User-Defined Registers

|  |  |  |
| --- | --- | --- |
| Register | Name | Role |
| $0 | $zero | Zero register that holds all zeros |
| $1 | $v0 | Function return, optional third argument, and used in pseudo-instructions. |
| $2 | $a0 | First Argument |
| $3 | $a1 | Second Argument |
| $4 | $t0 | Temporary register |
| $5 | $t1 | Temporary register |
| $6 | $s0 | Saved register |
| $7 | $lr | Link register |

Register $0 is the zero register, and it makes it simple to move values or initialize registers. It also is necessary whenever using a conditional branch with SLT; zero is always the value to which the register is being compared in the branch. Register $1 is defined to be the function return register, the register used in pseudo-instructions, and an optional third argument. Functions will place the return value into $v0, and a calling function can optionally use it as a third argument. Registers $2 and $3 are the argument registers $a0 and $a1. Whenever a function is called, parameters are passed to the function using these two registers. Registers $4 and $5 are temporary registers $t0 and $t1. They can be used and modified by any function. Register $6 is a saved register $s0, it can only be modified by the top-level function. If any other function uses it, it must be saved to memory first, and reloaded before the function returns. The last register, register $7, is the link register $lr. It is the register where the return address is stored during a function call. JL automatically stores the current $pc value into $7 whenever there is a jump to a function.

***Assignment Operations:***

***C Construct:***

***a = b + c;***

Let’s assume that the compiler has associated the variables a, b and c with the registers $s0, $t0 and $t1

*Assembly Code:*

ADD $s0, $t0, $t1

***C Construct:***

***a = b - c;***

Let’s assume that the compiler has associated the variables a, b and c with the registers $s0, $t0 and $t1

*Assembly Code:*

SUB $s0, $t0, $t1

***C Construct:***

***h[12] = g + h[8];***

Let’s assume that the compiler has associated the variables g with the registers $t0 and $t1 holds the base address of array h.

*Assembly Code:*

LW $s0,8($t1)

ADD $s0, $s0, $t0

SW $s0,12($t1)

***C Construct:***

***a = b & c;***

Let’s assume that the compiler has associated the variables a, b and c with the registers $s0, $t0 and $t1

*Assembly Code:*

AND $s0, $t0, $t1

***C Construct:***

***a = b | c;***

Let’s assume that the compiler has associated the variables a, b and c with the registers $s0, $t0 and $t1

*Assembly Code:*

OR $s0, $t0, $t1

***Control Flow Structures:***

***Some useful pseudo-instructions:***

Branch Less Than: BLT $t0, $t1, Label #if $t0<$t1, branch to Label

SLT $v0, $t0, $t1

BNE $v0, $zero, Label

Branch Greater Than: BGT $t0, $t1, Label #if $t0>$t1, branch to Label

SLT $v0, $t1, $t0

BNE $v0, $zero, Label

Branch Greater or Equal: BGE $t0, $t1, Label #if $t0>=$t1, branch to Label

SLT $v0, $t0, $t1

BEQ $v0, $zero, Label

Branch Less or Equal: BLE $t0, $t1, Label #if $t0<=$t1, branch to Label

SLT $v0, $t1, $t0

BEQ $v0, $zero, Label

***C Construct:***

***if (a CON b){ // CON may be ==,!=,>,<,>=,<=//***

***c = a + b;***

***} else {***

***c = a - b;***

***}***

Let’s assume that the compiler has associated the variables a, b and c with the registers $s0, $t0 and $t1

*Assembly Code:*

COND $a, $b, Label #COND 🡪 BEQ(==), BNE(!=), BGT(>), BLT(<), BGE(>=), BLE(<=)

SUB $t1, $s0, $t0

J Exit

Label: ADD $t1, $s0, $t0

Exit:

***C Construct:***

***int temp = 0;***

***for (i = 0, i CON 50, i++) { // CON may be ==,!=,>,<,>=,<=//***

***temp+= i;***

***}***

Let’s assume that the compiler has associated the variables i and temp with the registers $t1, $t0

*Assembly Code:*

ADDI $s0, $zero, 31

ADDI $s0, $s0, 19

ADD $t0, $zero, $zero

ADD $t1, $zero, $zero

Loop: COND $t1, $s0, Label #COND 🡪 BEQ(==), BNE(!=), BGT(>), BLT(<), BGE(>=), BLE(<=)

ADD $t0, $t0, $t1

ADDI $t1,$t1,1

J Loop

Label:

***C Construct:***

***int temp = 0;***

***int i = 0;***

***while (A[i] CON i) { // CON may be ==,!=,>,<,>=,<=//***

***temp+= A[i];***

***i++;***

***}***

Let’s assume that the compiler has associated the variables i and temp with the registers $t1, $t0 and $s0 has the base address of array A.

*Assembly Code:*

ADD $t0, $zero, $zero

ADD $t1, $zero, $zero

Loop: ADD $a0, $s0,$t1

LW $a0, 0($a0)

COND $a0, $t1, Label #COND 🡪 BEQ(==), BNE(!=), BGT(>), BLT(<), BGE(>=), BLE(<=)

ADD $t0, $t0, $a0

ADDI $t1, $t1,1

J Loop

Label:

***C Construct:***

***void main() {***

***int a[10];***

***int b = func(a, 10);***

***}***

***int func (int\* array, int size) {***

***int i, sum;***

***for (i = 0, sum = 0; i < size; i++){***

***sum += array[i];***

***}***

***return sum;***

***}***

Let’s assume that the compiler has associated the variable b with register $t0 and the address of array a is stored in $s0.

*Assembly Code:*

main: ADD $a0, $zero, $s0

ADDI $a1, $zero, 10

ADDI $v0, $zero, 2 #a0 is the first argument register

JL func #Jump and link by setting link register

ADD $t0, $zero, $v0 #a0 is the return register.

HALT

func: ADD $t0, $zero, $zero

ADD $t1, $zero, $zero

Loop: BGE $t0, $a1, Exit

ADD $v0, $a0, $t0

LW $v0, 0($v0)

ADD $t1, $t1, $v0

ADDI $t0, $t0, 1

J Loop

Exit: ADD $v0, $zero, $t1

JR $lr #Jump to the address stored in the link register