Computer Organization Project

Done by:

Houssam El dine Khaled

Sadek Mehiedinne

Abdallah Tourbah

Amir El Chakif

**Design Phase I:**

1. **Question 1:**

RISC Definition:

The RISC represents “Reduced Instruction Set Computer” which make hardware simpler by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations, for example load and store command will store and load the data; the processor reduce the cycles per instruction at the cost of the number of instructions per program.

CISC Definition:

The CISC represents **“**Complex Instruction Set Computer**”,** it comprises a complex instruction set where a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it’s complex.

|  |  |  |
| --- | --- | --- |
|  | RISC | CISC |
| Cycles Per Instructions | Reduce the cycles per instruction at the cost of the number of instructions per program.  (CPI = 1) | Attempts to minimize the number of instructions per program but at the cost of an increase in the number of cycles per instruction.  (CPI > 1) |
| Uses | Focus on software | Focus on hardware |
| Size of Instructions | Fixed sized instructions | Variable size instructions |
| Number of Registers | Requires more number of registers | Requires less number of registers. |
| Code Size | Large code size | Small code size |
| Word Size | One word size | More than one word size |

1. **Question 2:**
2. Instructions supported:

The Assembly code supports 4 types of instructions:

1. Arithmetic instructions comprising of the following instruction sets:
   1. add(for addition)
   2. sub(for subtraction)
   3. mul(for multiplication)
   4. div(for division)
   5. and(and logical operation)
   6. or(or logical operation)
   7. not(not logical operation)
2. Arithmetic Immediate instructions comprising of the same operations as the arithmetic used for constants,note the not logical operation does not have an arithmetic immediate operation. Also to note that when division occurs the result is rounded to the nearest value:
   1. addi (addition immediate with a constant)
   2. subi (subtraction immediate with a constant)
   3. muli (multiplication immediate with a constant)
   4. divi (division immediate with a constant)
   5. andi (and immediate logical operation with a constant)
   6. ori (or immediate logical operation with a constant)
3. Memory based instructions (Load and Store):
   1. Load (tranfers from main memory to registers)
   2. Store (tranfers from registers to main memory)
4. Branch instructions (beq, bne, bgt, and blt)
   1. beq (branch equal branches to a statement if condition is true)
   2. bneq (branch not equal branches to a statement if condition is true)
   3. bgt (branch greater than branches to a statement if condition is true)
   4. blt (branch less than branches to a statement if condition true)
5. Register operation (sll, srl, and init)
   1. sll (shift left logical shift to the left)
   2. srl (shift right logical shifts to the right)
   3. init(initialization used to initialize registers to a certain constant)
6. Instruction fields:

As for the instruction structures, we managed to design 2 instruction fields them being:

ABM-Format supporting Arithmetic, branch, and memory instructions and comprises of 32 bits divided in the following manner:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Opcode 1 | R1 6 | R2 6 | Rd 6 | Function 6 | Branch address 6 | Memory 1 |

31 0

IR-Format supports Arithmetic immediate instructions, initialization, as well as shifting operations and comprises of 32 bits:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode 1 | R1 6 | Rd 6 | Constant 15 | Function 4 |

31 0

1. Explanation of the fields:

Since the processor used is the RISC, the number of bits for all formats used should be the same. In this language, 32 bits have been used for each format. In this case, we managed to make 2 formats, the ABM format and the IR format. The ABM format is used for arithmetic operations, branch based operations, and memory operations. The IR format on the other hand supports arithmetic immediate operations as well as initialization and shifting.

Both formats contain an opcode of 1 bit since we have two formats. An opcode of bit 0 will refer to the ABM format while that of bit 1 will refer to the IR format. In addition to the opcode both contain R1 and Rd, keeping in mind the ABM format contains R2, which signify source (R1 and R2) and destination (Rd) registers. They contain 6 bits since 64 registers of 32 bits were used.

For the ABM format, function, branch address, and memory slots have been established the former two containing 6 bits because we have 1 arithmetic instructions and 4 branch instructions and the latter containing 1 bit since we have two memory instructions. If the function and the branch address sections have a value of 0 in all 6 bits, then the memory section will be activated, a value of 0 signifying load and 1 signifying store. If the function section contains a value of 000001 and above, then one of the arithmetic operations will be executed provided the other two sections have a value of 0. Similarly, for the branch address section, if it has a value of 000001 and above then branch instructions will be executed provided the other two sections have a value of 0.

For the IR format, a 4 bits function section signifies which instruction to be used be it arithmetic immediate instructions, from 0000 to 0101, as well as shifting and initialization operations, from 0110 to 1000. As for the constant section it signifies what we are initializing, shifting by how much, or with what constant we can be adding our variable with. It contains 15 bits since the numbers supported by the system range between - 2^ 14 till 2^14 – 1. Initialization operation only takes constant therefore the R1 section has a value of 0.

1. **Question 3:**

|  |  |  |
| --- | --- | --- |
| Type | Syntax | Corresponding opcode |
| Register operation | init \*zer ,const , \*rd | 1 |
| Arithmetic Immediate | addi \*r1 , const, \*rd | 1 |
| Branch operation | bne \*r1, \*r2, <branch> | 0 |
| Memory Access | load \*r1, \*r2, \*rd | 0 |
| Arithmetic operation | sub \*r1, \*r2, \*rd | 0 |
| Arithmetic operation | not \*r1, \*rd | 0 |
| Arithmetic operation | Add \*r1, \*r2, \*rd | 0 |
| Arithmetic operation | Sub \*r1, \*r2, \*rd | 0 |
| Arithmetic operation | Or \*r1, \*r2, \*rd | 0 |
| Arithmetic operation | Mul \*r1, \*r2, \*rd | 0 |
| Arithmetic operation | Div \*r1, \*r2, \*rd | 0 |
| Arithmetic Immediate | subi \*r1 , const, \*rd | 1 |
| Arithmetic Immediate | muli \*r1 , const, \*rd | 1 |
| Arithmetic Immediate | divi\*r1 , const, \*rd | 1 |
| Arithmetic Immediate | ori \*r1 , const, \*rd | 1 |
| Arithmetic Immediate | andi \*r1 , const, \*rd | 1 |
| Memory Access | store \*r1, r2, \*rd | 1 |
| Branch operation | beq \*r1, \*r2, <branch> | 0 |
| Register operation | sll const, \*r1, \*rd | 1 |
| Register operation | srl const , \*r1, \*rd | 1 |
| Branch operation | blt \*r1, \*r2, <branch> | 0 |
| Branch operation | bgt \*r1, \*r2, <branch> | 0 |

**4) Question 4:**

**Register (Name, Number, Use):**

|  |  |  |
| --- | --- | --- |
| Name | Number | Use |
| \*zer | 0 | Constant value zero mainly used for initialization |
| \*sor0-\*sor29 | 1-31 | Source registers |
| \*tep0-\*tep29 | 32-62 | Temporary registers |
| \*stk | 63 | Stack pointer |

**5) Question 5:**

|  |  |
| --- | --- |
| Full code(divided according to its lines) | machine language(for each line of code) |
| init \*zer, 2, \*sor0 | 10000000000010000000000000100110 |
| Addi \*zer, 3, \*sor1 | 10000000000100000000000000110000 |
| Bne \*sor0,\*sor1,<qr> | 00000010000100000000010010000010 |
| <qr>: Load \*sor3, \*sor0,\*tep0 | 01000100100000000000000000000000 |
| Sub \*sor1, \*sor0, \*tep1 | 000001000000110000100000100000000 |
| Not \*tep1, \*tep8 | 01000010000001010000001110000000 |
| Add \*sor0,\*sor3, \*tep2 | 00000010001001000100000010000000 |
| Store \*sor3, \*sor0, \*tep0 | 00001000000011000000000000000001 |
| Bgt \*tep1, \*tep8, <rot> | 01000011010000000000010110000010 |
| <rot>:And\*tep1,\*tep8, \*sor5 | 01000011010000001100001010000000 |
| Or \*tep1, \*tep8, \*sor9 | 01000011010000010100001100000000 |
| Subi \*sor9, 1, \*sor13 | 10010100011100000000000000010001 |
| Divi \*sor13, 2, \*tep17 | 10011101011110000000000000100011 |
| Blt \*sor9, \*sor5, <w> | 00010100001010000000010100001000 |
| Andi \*sor9, 1, \*tep19 | 00010101100110000000000000010101 |
| Ori \*sor5, 0, \*tep20 | 00001101101000000000000000000100 |
| Div \*sor9, \*sor13, \*sor21 | 00010100011100101100001000000000 |
| <w>: Sll \*sor9, 2, \*tep15 | 10010101011110000000000000100111 |
| Muli \*tep15, 4, \*sor11 | 11011110011000000000000001000010 |
| Beq \*sor9, \*tep1, <dot> | 00010101000010000000010000000100 |
| Srl \*tep15, 2, \*tep26 | 11011111110100000000000000101000 |
| <dot>:Mul\*sor0,\*tep15, \*tep29 | 00000011011111111100000110000000 |

**6) Question 6:**

Since our assembly code is byte addressable and contains 32 bits of machine language which is equivalent to 4 bytes, and since in RISC all instruction formats store the same number of bits therefore we increment by 4 for every instruction.

**7) Question 7:**

**Reference data:**

Function: Hex(Opcode/function or memory access/Load or Store):

Add 0/1/0 Format: ABM

Sub 0/2/0 ABM

Mul 0/3/0 ABM

Div 0/4/0 ABM

And 0/5/0 ABM

Or 0/6/0 ABM

Not 0/7/0 ABM

Addi 1/0 IR

Subi 1/1 IR

Muli 1/2 IR

Divi 1/3 IR

Ori 1/4 IR

Andi 1/5 IR

Sll 1/7 IR

Srl 1/8 IR

Init 1/6 IR

Beq 0/8/0 ABM

Bne 0/9/0 ABM

Blt 0/10/0 ABM

Bgt 0/11/0 ABM

Load 0/0/0 IR

Store 0/0/1 IR

**Basic Instructions Format:**

ABM-Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Opcode 1 | R1 6 | R2 6 | Rd 6 | Function 6 | Branch address 6 | Memory 1 |

31 0

IR-Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode 1 | R1 6 | Rd 6 | Constant 15 | Function 4 |

31 0

|  |  |  |
| --- | --- | --- |
| Name | Number | Use |
| \*zer | 0 | Constant value zero mainly used for initialization |
| \*sor0-\*sor29 | 1-31 | Source registers |
| \*tep0-\*tep29 | 32-62 | Temporary registers |
| \*stk | 63 | Stack pointer |

**Design Phase II:**

After changing minor issues to our old assembly code in phase 1:

|  |
| --- |
| Full code(divided according to its lines) Assembly code |
| init \*zer, 2, \*sor0 |
| Addi \*zer, 3, \*sor1 |
| Bne \*sor0,\*sor1,<qr> |
| Not \*tep1, \*tep8 |
| Sub \*sor1, \*sor0, \*tep1 |
| <qr>: init \*zer , 4, sor2 |
| Load \*sor3, \*sor0,\*tep0 |
| Add \*sor0,\*sor2, \*tep0 |
| Store \*sor3, \*sor2, \*tep0 |
| Bgt \*tep1, \*tep8, <rot> |
| <rot>:And\*tep1,\*tep8, \*sor5 |
| Or \*tep1, \*tep8, \*sor9 |
| Subi \*sor9, 1, \*sor13 |
| Divi \*sor13, 2, \*tep17 |
| Blt \*sor9, \*sor5, <w> |
| Andi \*sor9, 1, \*tep19 |
| Ori \*sor5, 1, \*tep20 |
| Div \*tep19, \*tep20, \*sor21 |
| <w>: Sll \*sor3, 2, \*tep15 |
| Muli \*tep15, 4, \*sor11 |
| Beq \*sor11, \*tep1, <dot> |
| Srl \*sor3, 2, \*tep26 |
| <dot>:Mul\*sor0,\*tep15, \*tep29 |

Sor3 is Array register address

|  |
| --- |
| Full C code |
| a=2 |
| b=3+0 |
| if (a==b){ |
| t1=b-a |
| t8=t1’} |
| Else{c=4 |
| t2=c+a |
| A[c]=t2+A[a] } |
| if(t1>t8){ |
| e=t1&&t8 |
| i=t1||t8 |
| m=i-1 |
| t17=m/2} |
| if(i>e){ |
| t19=i&&1 |
| t20=e||1 |
| u=t19/t20} |
| Else{ |
| t15= A index\*2 |
| k=t15\*4} |
| If(k!=t1){ |
| t26=A index/2} |
| Else{t29=a\*t15} |