**Chapter 1: Computer Abstraction & Technology**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Register Number | Register Usage | Preserved? |
| x0 | 0 | Constant (0) |  |
| x1 (ra) | 1 | Return Address | Y |
| x2 (sp) | 2 | Stack Pointer | Y |
| x3 (gp) | 3 | Global Pointer | Y |
| x4 (tp) | 4 | Threat Pointer | Y |
| x5 – x7 | 5-7 | Temporaries | N |
| x8 – x9 | 8-9 | Saved | Y |
| x10 – x17 | 10-17 | Arguments/Results | N |
| x18 – x27 | 18-27 | Saved | Y |
| x28 – x31 | 28-31 | Temporaries | N |

*Equations*  *RISC-V Instruction Types*

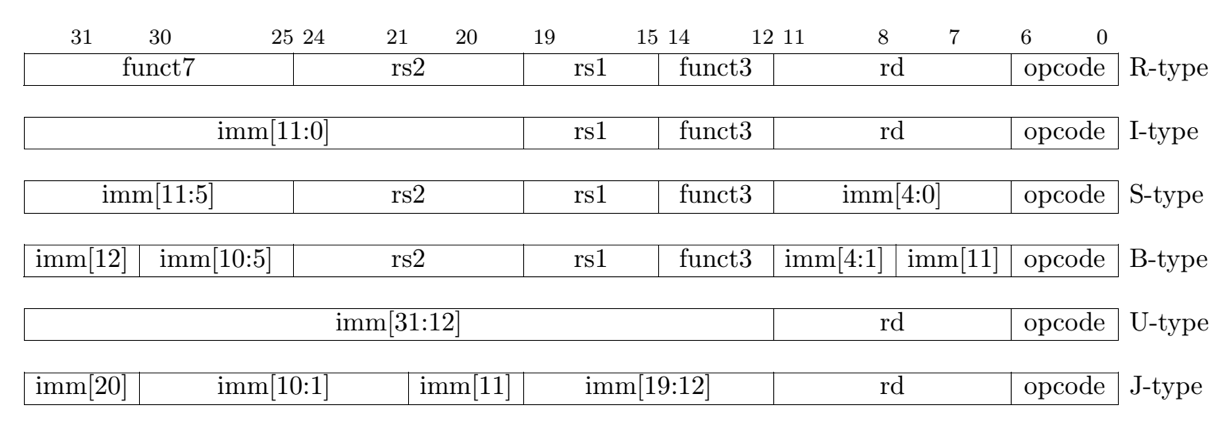
CPU Execution Time = CPU Clock Cycles \* Clock Cycle Time OR = CPU Clock Cycles / Clock Cycle Rate

CPU Time = Instruction Count \* CPI \* Clock Cycle Time OR = Instruction Count \* CPI / Clock Cycle Rate

MIPS = Instruction Count / Execution Time \* 10^6 OR = Clock Cycle Rate / CPI \* 10^6

*Negative Decimal to Binary Conversion*

2 = 00000000 00000000 00000000 00000000 00000000 00000000 00000010

-2 = 11111111 11111111 11111111 11111111 11111111 11111111 11111110 *RISC-V Instruction Types (below)*

**Chapter 2: Instructions – Language of the Computer**

*PC Incrementation*

For branch instructions, the program counter is incremented by the branch offset instead of 4.

PC = register + 4 OR = register + branch offset

*RISC-V Addressing Modes*

Graphical user interface

Description automatically generated with low confidenceAddressing Modes: RISC-V standard for addressing for various instructions.

* Immediate Addressing (I-Type): operand is a constant within the instruction
* Register Addressing (R-Type): operand is a register
* Base/Displacement Addressing (S/SB-Type): operand is at the memory location found from address from sum of a register & constant within instruction
* PC-Relative Addressing (U/UJ-Type): branch address is the sum of the PC & constant within instruction

*RISC-V Addressing Modes (right)*

Timeline

Description automatically generated*Compiling & Running a Program*

Compiler: transforms a C program into an assembly language program (in this case, RISC-V). This reduces the amount of code as assembly instructions are higher-level

Assembler: converts accepted instructions into a more simplified translation (psuedoinstruction)

Ex. mv & li instructions in assembly language are equivalent to addi in machine language

Linker: links together easy assembly machine language program into one, outputting an executable file that can be run on a computer

Step 1. Place code & data modules symbolically in memory

Step 2. Determine addresses of data & instruction labels

Step 3. Path internal & external references

*RISC-V & MIPS Instruction Formats (above)*

Table

Description automatically generatedLoader: operating system reads executable into memory& starts the program

Step 1. Read executable header to determine size of text & segments

Step 2: Create address space for text & data based on size

Step 3: Copy instruction & data from the executable file into memory

Step 4: Copy parameters to main program on the stack

Step 5: Initialize processor registers and set stack pointer to first free memory location

Step 6: Branch to start-up routine. Start-up routine copies parameters into argument registers,   
calls main routine, & terminates once

***DO NOT Forget!!***

* **Powerful instructions DO NOT mean higher performance**
* **Writing in assembly language DOES NOT always obtain the highest performance**
* **Sequential word/doubleword addresses in machines with byte addressing DO NOT differ by one**
* **DO NOT use a pointer to an automatic variable outside its defining procedure.**

routine has returned using an

exit system call

**Chapter 3: Arithmetic for Computers**

*Basic Arithmetic*

* Binary addition: add from right to left

*Example: Add two binary values b1 and b2 to produce the sum b3. c is the current bit being access in each binary value.*

* 1. *Diagram

     Description automatically generated*If b1(c) + b2(c) = (1+1)
     + b3(c) = 0
     + b1(c+1) = b1(c+1) + 1
  2. Else if b1(c) + b2(c) = (0+1)
     + b3(c) = 1
     + b1(c+1) = b1(c+1) + 0
  3. Else //Else if b1(c) + b2(c) = (0+0)
     + b3(c) = 0
     + b1(c + 1) = b1(c + 1) + 0

*Bit Masking*

Mask: define bits to clear and bits to keep by performing a bitwise OR operation to set the masked bits and using a bitwise XOR operation to change already masked bits.

*Multiplication*

* Initial Values: b1 *multiplicand* & b2 *multiplier*
* b3 = 0 *product*
* c = 0 *bit counter*

**\*n o t e\***

M O O R E ‘ S L A W

Determine if operand is to be added before beginning multiplication.

Saves time by avoiding looking at all 64 bits but instead only those being adding to.

* if b2 = 1

*product = multiplicand \* multiplier*

* + *b3 = b1 \* b2*

*productReg = product + multiplicand*

* + b3 = b3 + b1
* slli b1, b1, 1

*shift the multiplicand register 1 bit left*

* srli b2, b2, 1

*shift the multiplier register 1 bit right*

* c += 1

*iterate through all 64 bits*

* if(c == 64)

*once all bits have been multiplied, exit*

* + return b3 exit program

*Diagram

Description automatically generatedDivision*

* *Initial Values: r remainder & d1 dividend & d2 divisor*
  + li r, 0
  + li d1, 9
  + li d2, 2
  + li d28, 0 *//constant*
  + li d29, 1 *//constant*
  + li d30, 0
  + li d31, 0 *//counter*
* *q quotient* 
  + li q, 0
* *subtract the divisor register from the remainder register*
* *place result in remainder register*
  + sub r, r, d2
* *check if the remainder is 0 using XOR*
  + xor d30, d2, 0
* *branch to remainder-left subroutine if d2 is equal to 1*
  + beq d2, d29, remainder-left
* *pass through if d2 is equal to 0*
* *qrestore original value by adding divisor register to remainder register* 
  + add r, d2, r
* *shift the quotient register 1 bit left*
* *Set least significant bit to 0* 
  + slli q, q, 1
* *shift the divisor register 1 bit right* 
  + srli d2, d2, 1
* *add 1 to bit counter*
  + add d31, d31, d29
* *branch if all 64 bits have been divided*
  + beq d31, d5, pass-routine
* remainder-left:
* *shift the quotient register left*
* slli q, q, 1
* *set least significant bit of quotient register to 0*
* xor tempR, q, 0
* beq tempR, d28, set-bit
* j divide
* set-bit:
* addi d30, d30, 1
* j remainder-left