RICHARD ROBINSON

ASSEMBLY LANGUAGE

Introduction

Performance Metrics

There are several metrics of performance of programs by processors. These include

- *Execution time*, the total time required for the computer to complete a task (also called response time).
- *Bandwidth*, the number of tasks completed per unit time (also called throughput).
- *Performance*, the inverse of execution time.
- *Clock cycle*, the time for one (processor) clock period, which runs at a constant rate.

Consequently, the CPU time is defined as

$$t_{\text{cpu}} = n_{\text{cycles}}/r_{\text{clock}} \quad \text{for} \quad n_{\text{cycles}} = I \times \text{CPI} \tag{1}$$

CPI: the average number of clock cycles per instruction for a program.

The MIPS unit is another performance metric, given by

$$\text{MIPS} = I/t_{\text{exec.}} \cdot 10^6 = r_{\text{clock}}/\text{CPI} \cdot 10^6 \tag{2}$$

MIPS: A measurement of program execution speed based on the number of millions of instructions.

RISC-V Syntax

RISC-V is an assembly language with a basic syntax of

```
add x1,x2,x3        // x1 = x2 + x3
```

The operands `x0-x31` are the 32 registers with `x0` always equaling `0`. Data must be in registers for arithmetic. Additionally, there are 2^{61} memory words `Memory[8i]` which may be accessed only by data transfer instructions.

Word: Half of a doubleword; a group of 32 bits.

In RISC-V, a doubleword is a group of 64 bits. There are never any variables in RISC-V; all variables are converted into respective registers. For example, if `g` and `h` are `x20,x21` and the base address is `x22` of array `A`, then the C code

Address: The index for a data element in a memory array.

```
A[z] = h + A[y]
```

in RISC-V is translated to

```
ld  x9,8y(x22)      // temp reg x9 gets A[y]
add x9,x21,x9       // temp reg x9 gets h + A[y]
sd  x9,8z(x22)      // stores x9 into A[z]
```

The reg added to form the address (x22) is the *base reg* and the DTI constant (64) is the *offset*. To get the proper byte address, the offset to be added to the base reg is $8c$ where c is the initial constant.

Byte: A group of 8 bits; there are 8 bytes in a doubleword.

This implies that the byte address of an element is a multiple of 8; that is, element i has byte address $8i$ where the first element is i_0.

Binary & Hexadecimal

Binary to decimal base is calculated via

$$\sum x_i 2^i \tag{3}$$

where x_i is the ith digit starting at $i = 0$ and increasing leftwards. In RISC-V, the least or most sig. bit is the rightmost or leftmost bit, respectively.

Two's complement representation for signed binary numbers assigns leading 0s as positive and leading 1s as negative. The complements are related by

$$\overline{x} + 1 = -x \tag{4}$$

Binary is used in the instruction format, which represents an instruction in 6 fields, typically the *R*-type:

1. `opcode`: basic op of the instruction (7b).
2. `rd`: reg dest operand; gets result of op (5b).
3. `funct3/7`: additional opcodes (3/7b).
4. `rs2/1`: second / first reg source operand (5b).

In Hexadecimal, all groups of 4 binary digits may be represented by a single hexadecimal digit, ranging from $0-9, a-f$ such that

```
9 = 1001    a = 1010    f = 1111
```

Logical & Conditional Ops

The typical logical operators occur in RISC-V, as do:

1. Shift left / right (`sll` / `srl`): moves all bits in a doubleword left / right, filling emptied bits with 0s. Shifting left i bits is equal to multiplying by 2^i.
2. Bitwise XOR (`xor`): an operation which returns 1 only if the values differe in the two operands.
3. Bitwise NOT (`xori`): inverts the bits in a doubleword; equivalent to XOR with one operand being `11...1`.

For conditional statements, the commands `bne` and `beq` are used. For example,

```
if (i == j) f = g + h; else f = g - h
```

in C where `f-g` are `x19-x23`, the equivalent RISC-V code is

```
      bne x22, x23, Else     // goto Else if i~=j
      add x19, x20, x21      // f=g+h if i == j
      beq x0,   x0, Exit     // goto Exit
Else: sub x19, x20, x21      // f=g-h if i~=j
Exit:
```

Likewise, for loops, the same commands are used in addition to the label `Loop:`. For example,

```
while (save[i] == k) i++
```

in C where `i,k,save` are `x22,x24,x25`, the code is

```
Loop: sll x10, x22, 3      // treg x10=8i
      add x10, x10, x25    // x10 = save[i]
      ld x9, 0(x10)        // treg x9 = save[i]
      bne x9. x24, Exit    // goto Exit if
                           // save[i]~=k
      add x22, x22, 1      // i++
      beq x0, x0, Loop     // goto Loop
Exit:
```

This can be extended to the comparison `0 <= x < y`, which is valid for the index out-of-bounds check. For example, the check `x20 >= x11 || x20 < 0` is given by `bgey x20, x11, OutOfBounds`.

RISC-V Operation