CSM CS61C Note #2: RISC-V Intro

Anthony Han

1 Assembly Variables: Registers

- Assembly does not have variables – operands are objects called **registers**.
- Registers have **no type** – operation determines how register contents are interpreted.
- In RV32 variant of RISC-V:
  - 32 registers (x0 - x31), each of which is 32b wide.
  - **x0 hard coded to be 0**; any write to this register is ignored.
  - An additional register, **program counter (PC)**, that stores the address of the current instruction being executed.

2 RISC-V Instructions

In RV32, instructions are **32b long**. They must be word aligned, or half-word aligned if 16b instructions are enabled. For now, we can classify them into these categories:

Register Instructions

A register instruction takes in two **source registers**, do some computation, and write the result into the **destination register**.

- Syntax:

```
1 add x1, x2, x3 # a = b + c
2     # add - operation code (opcode) for addition
3     # x1 - destination register (rd)
4     # x2, x3 - source registers (rs)
5     # # is assembly comment syntax
```

- Here are all the register instructions:
  - Arithmetic operations: add, sub, mul, div.
  - Logical operations: and, or, xor, sll, srl, sra.
  - Conditional set: slt, sltu.
- We shall note that although both srl and sra do a right shift, there is a difference:
  - Shift right logical (srl) **0-extends**; that is, after taking off the rightmost bits, we **insert 0s on the left**.
  - Shift right arithmetic (sra) **sign-extends**; that is, after taking off the rightmost bits, **we insert the leftmost bit of original number on the left.** In fact, sra by n bits gives you the result of the integer being floor divided by 2^n.

Immediate Instructions

Just like programming in C, we need numerical constants. In RISC-V, you can do arithmetic operations between a register and an immediate using an immediate instruction.

- For most operations, to get the opcode for the immediate instruction, we can **simply append an i** to the opcode of the corresponding register instruction. However, there is **no such thing as subi, muli, divi**.
- Syntax:

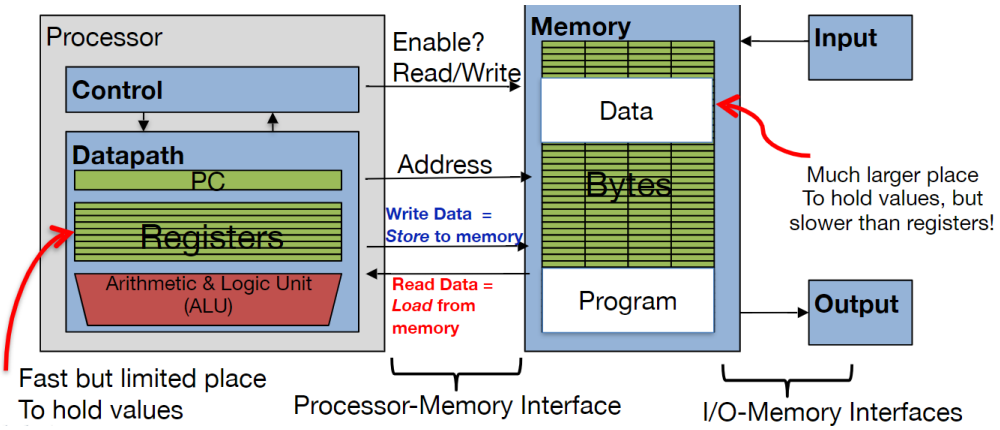
```
addi x1, x2, 1 # a = b + 1
    # addi - operation code (opcode) for immediate addition
    # x1 - destination register (rd)
    # x2 - source register (rs)
    # 1 - immediate
```

- **Immediates are necessarily small**; for immediate instructions, it is a 12b signed integer. Since **in two's complement, sign extension does not change the value of a signed integer**, RISC-V would have sign-extended the immediate before any arithmetic operation is done.
- To do arithmetic operations on two immediates, first load one of them onto a register, then use an immediate instruction.

```
# To do 123 + 345, we write ...
li   x1, 123        # Equivalent to addi x1, x0, 123
addi x2, x1, 345    # Do the actual add

# Similarly, since subi is not a valid instruction,
# To get something like subi x2, x1, 5, we write ...
li   x3, 5          # Equivalent to addi x3, x0, 5
sub  x2, x1, x3     # Do the sub using register instruction
```

Memory Instructions



In RISC-V, the only way to access memory is through load and store instructions. **The little endian convention** is preserved; that is, **the address of the word is the same as that of the least significant byte.**

- To transfer data from memory to register, we use **load word (lw)**. For example, the following code segment

```
int A[100];
g = h + A[3];
```

  can be written in RISC-V as

```
# Assume that reg x13 stores the address of A
lw  x10, 12 (x13) # Reg x10 gets M[R[x13] + 12]
    # x13 - base register (pointer to A[0])
    # 12 - offset in bytes; must be constant at assembly time
add x11, x12, x10 # g = h + A[3]
```

- To transfer data from register to memory, we use **store word (sw)**. As an example, the following code segment

```
int  A[100];
A[10] = h + A[3];
```

  can be written in RISC-V as

```
lw  x10, 12 (x13) # Temp reg x10 gets A[3]
add x10, x12, x10 # Temp reg x10 gets h + A[3]
sw  x10, 40 (x13) # A[10] = h + A[3]
    # x13 - base register (pointer to A[0])
    # 12, 40 - offset in bytes
```

- Instead of tuning the offset, we can instead change the address stored in the base register. For example, code segment

```
int A[100];   /* x13 */
int i;        /* x14 */
g = h + A[i]; /* h = x12, g = x11, tmp = x15 */
```

  can be written in RISC-V as

```
slli x15, x14, 2    # Multiply i by 4
add  x15, x15, x13  # Get memory location
lw   x10, 0 (x15)   # Load the integer on that location
add  x11, x12, x10
```

- RISC-V also has byte/half word data transfers:
  - **lb/lh** for load byte/half word, **sb/sh** for store byte/half word.
  - **lb/lh automatically sign extends!**

```
lb x10, 3 (x11)
    # Load the third byte from contents of x11
    # Automatic sign extension
        # MSB of the byte copied to upper bits in x10
    # x11 - base register (a pointer)
```

  - **lbu/lhu** for load unsigned byte/half word to avoid sign extension.
  - *sbu/shu* is not necessary, since the number of bytes we write equals the number of bytes received.

- Although they look similar, **load address (la) and load word (lw) are very different.**

```
.data
n: .word 9
.text
la t0, n
    # Load address takes in a label n...
    # And write the "address of the label" to t0
    # Used to create pointers pointing to function, data, etc.
    # Pseudo-instruction that turns into auipc + addi
lw t0, 0(t0)
    # Add the offset 0 to the value stored at the base register t0...
    # And read the memory at that memory location
    # Used to read from memory
    # Proper RISC-V instruction
```

Branch Instructions

In RISC-V, we can change the control flow with conditional branches, which takes in two source registers and a label; if the two source registers satisfy the specified relation, jump to the label.

- **Branch if equal (beq)** or **branch if not equal (bne).** If we have C code

  ```
  /* Assume f = x10, g = x11, h = x12, i = x13, j = x14. */
  if (i == j)
      f = g + h;
  else
      f = g - h;
  ```

  We can convert this into RISC-V:

  ```
  # If-else-block
      beq x13, x14, else
      add x10, x11, x12
      j exit
  else: sub x10, x11, x12
  exit:
  ```

- **Branch if less than (blt)** or **branch if greater than or equal (bge)** - both **have unsigned versions**.

  ```
  blt  reg1, reg2, label
      # If reg1 less than reg2, goto label
      # Registers are interpreted as signed integers
  bltu reg1, reg2, label
      # If reg1 less than reg2, goto label
      # Registers are interpreted as unsigned integers
  # Similar syntax for bge and bgeu
  bge  reg1, reg2, label
  bgeu reg1, reg2, label
  ```

- No such thing as *bgt* in RISC-V - **switch argument instead**. We have pseudo-instructions, where the assembler does the conversion.

  ```
  bgt x2, x3, foo
  # Becomes blt x3, x2, foo
  ```

- As in the case of immediate instructions, if you want to compare a register with an immediate, load the immediate onto a register and then do the branching.

Jump Instructions

In addition to conditional branches, in RISC-V, we can also use the unconditional jump instructions. We only have two actual unconditional branching instructions:

- **Jump and Link (jal)** - Useful for continuing a loop/calling another function.

```
jal rd, label
    # At machine code level, add immediate value to
    # program counter (PC), then go to that location
        # Offset is 20b, sign extended and left-shifted one
    # Store into rd the value of PC + 4
        # So it knows where to return to
# Nice, easy shortcut
jal label # == jal ra, label
# Pseudo-instruction if don't care where it returns to
j label   # == jal x0, label
```

- **Jump and Link Register (jalr)** - Useful for returning from function/calling pointer to function.

```
jalr rd, rs, offset
    # At machine code level, add immediate value to
    # source register (rs), then go to that location
        # Offset is 12b, sign extended
    # Store into rd the value of PC + 4
        # So it knows where to return to
# Nice, easy shortcut
jalr rs # == jalr ra, rs, 0
# Pseudo-instruction if don't care where it returns to
jr rs   # == jalr x0, rs, 0
# Pseudo-instruction to return from a function
ret     # == jr ra == jalr x0, ra, 0
```

3 Pseudo-Instructions

A pseudo-instruction is an assembly instruction that is to be replaced by the assembler. In RISC-V, some common pseudo-instructions are

```
nop
    # Pseudo-instruction that does nothing
    # Realized through add x0, x0, x0
    # Useful for halting the program
li rd, imm
    # Pseudo-instruction that loads a 32b immediate
    # Realized using lui and addi
mv rd, rs
    # Pseudo-instruction that moves value in one register to another
    # Realized through add rd, rs, x0
not rd, rs
    # Pseudo-instruction that does a bitwise not
    # Realized through xori rd, rs, -1
la rd, label
    # Pseudo-instruction that writes "the address of a label" to rd
    # Realized using auipc and addi
j label
    # Pseudo-instruction that jumps to a label without changing the ra
```

```
    # Realized through jal x0, label
jr rs
    # Pseudo-instruction that jumps to an address without changing the ra
    # Realized through jalr x0, rs, 0
ret
    # Pseudo-instruction that returns from a function without setting the ra
    # Realized through jalr x0, ra, 0
```