Live Coding Time (bitwise.s)

Assume that x (using s0) and y (using s1) are both integers. Assume further that integers are 4 bytes. Note that bits are numbered from *right to left* in a word starting with bit 0. Write two lines of RISC-V code to set x to the value contained in bits 15 to 10 of y.

```
srli s1, s1, 10       #  x = y>>10
andi s0, s1, 0x3F  # x = x&0x3F =(y >> 10) & 0x3F;
```

```
slli s1, s1, 16  # x = y <<16
srli s0, s1, 26 # x = x >> 26 (=(y<<16)>>26)
```

RARS: Pseudo instructions

- **Definition**
  - A RISC-V instruction that doesn't turn directly into a machine language instruction, but broken up into several other RISC-V instructions
  - Provided by an assembler but not implemented in hardware
- **Examples**
  - Load address: la t0, str
  - Load immediate: li t0, 0x40044005
  - Move register content: mv t1, t2 # copy t2 to t1

RARS: Pseudo Instructions

- **Assembler converts pseudoinstructions to real RISC-V instructions**
- **Example 1, move:**

  mv t1, t2 # copy t2 into t1

  becomes

  add t1, t2, zero
- **Example 2, load immediate**

li s1, 0xAABBCCDD **becomes**

lui s1, 0x000AABBD

addi s1, s1, 0xFFFFFCDD

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - ➔ Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

Making Decisions

- **To build a calculator**
  - instructions that manipulate data: arithmetic, memory access, logical and shift
- **To build a computer**
  - need instructions to make decisions
  - provide labels to support "`goto`" that jumps to places in code
    - Horrible style for C, but necessary for RISC-V

C Decisions: `if` Statements

- **2 kinds of `if` statements in C**
  - `if` (*condition*) *clause*
  - `if` (*condition*) *clause1* `else` *clause2*

- **Rearrange for RISC-V:**

```
    if  (condition) goto L1;
          clause2;
          goto L2;
    L1: clause1;
    L2:
```

Conditional Branch

- **Conditional branch instruction:**
  - `beq  register1, register2, L1`
  - `beq` is "Branch if (registers are) equal"
    Same meaning as (using C):
    `if  (register1==register2) goto L1`
- **Complementary decision instruction**
  - `bne  register1, register2, L1`
  - `bne` is "Branch if (registers are) not equal"
    Same meaning as (using C):
    `if  (register1!=register2) goto L1`

Unconditional Branch

- **Unconditional branch instruction**

  jal zero, label

  - jump (or branch) directly to the given label without needing to satisfy any condition
  - “zero” used as we won’t be returning
  - A label is a sequence of letters, numbers, underbars (_) and dots
- **Same as:**

  ```
  beq        zero,zero,label
  ```

  since it always satisfies the condition.
- **Same meaning as (using C):**

  ```
  goto label
  ```

Labels

- **Labels provided by programmer correspond to actual memory locations**
- **Labels get translated to locations embedded in instructions**
- **No memory used to store labels**

Memory Location	Instruction
5048	`add s0, x0, x0`
5052	`addi s1, s0, 1`
5056	`beq s1, s0, exit`
5060	
5064	
5068	
…	
6000(exit)	`ori …`

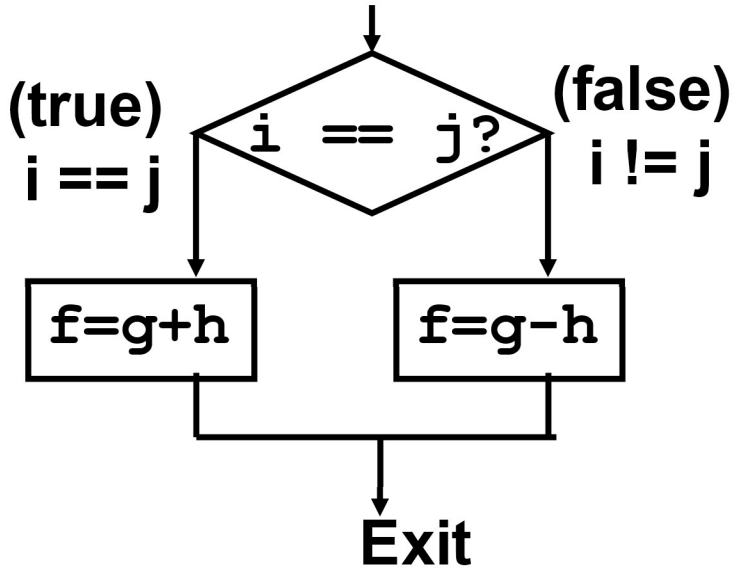
After `beq` evaluated, at which location does the instruction to be executed exist?

What if `beq` were replaced with `bne`?

Example

- **Compile**

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



- **Use this mapping:**

f: s0
g: s1
h: s2
i: s3
j: s4

Example: Answer

- **Compile**

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

```
f: s0   g: s1
h: s2   i: s3
j: s4
```

- **Final compiled RISC-V code:**

```
      beq s3,s4,True # branch i==j
      sub s0,s1,s2   # f=g-h(false)
      jal zero, Fin  # goto Fin
True: add s0,s1,s2   # f=g+h (true)
Fin:
```

- **Note: Compiler automatically creates labels to handle decisions (branches). Generally labels are not found in HLL code.**

Loops in C

- **Simple loop in C; `A[]` is an array of `ints`**

```
    do {
  g = g + A[i];
  i = i + j;
    } while (i != h);
```

- **Rewrite this as:**

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

- **Use this mapping:**

g	h	i	j	Base of A
s1	s2	s3	s4	s5

Loops in Assembly

g	h	i	j	Base of A
s1	s2	s3	s4	s5

- **Original code:**

```
Loop:g = g + A[i];
     i = i + j;
     if (i != h) goto Loop;
```

- **Final compiled RISC-V code:**

```
Loop: slli t1,s3,2   #t1= 4*i
      add t1,t1,s5   #t1=addr A
      lw  t1,0(t1)   #t1=A[i]
      add s1,s1,t1   #g=g+A[i]
      add s3,s3,s4 #i=i+j
      bne s3,s2,Loop # goto Loop
                     # if i!=h
```

Loops in C/Assembly

- **Three types of loops in C:**
  - `while`
  - `do... while`
  - `for`

- **Each can be rewritten as either of the other two, so the method used in the previous example can be applied to `while` and `for` loops as well.**

- **Key Concept: Though there are multiple ways of writing a loop in RISC-V, the key to decision making is conditional branch**

Switch Statement in C: Example

- Choose among four alternatives depending on whether `k` has the value 0, 1, 2 or 3. Compile this C code:

```
switch (k) {
 case 0: f=i+j; break; /* k=0 */
 case 1: f=g+h; break; /* k=1 */
 case 2: f=g-h; break; /* k=2 */
 case 3: f=i-j; break; /* k=3 */
}
```

Switch Statement in C: Rewrite

- **Rewrite it as a chain of if-else statements, which we already know how to compile:**

```
if(k==0) f=i+j;
  else if(k==1) f=g+h;
    else if(k==2) f=g-h;
      else if(k==3) f=i-j;
```

- **Use this mapping:**

```
f:s0, g:s1, h:s2,
i:s3, j:s4, k:s5
```

Switch Statement in RISC-V Code

- **Final compiled RISC-V code:**

```
     bne s5,0,L1     # branch k!=0
     add  s0,s3,s4 #k==0 so f=i+j
     jal zero,Exit          # end of case so Exit
L1:  addi t0,s5,-1   # t0=k-1
     bne  t0,0,L2    # branch k!=1
     add  s0,s1,s2 #k==1 so f=g+h
     jal zero,Exit          # end of case so Exit
L2:  addi t0,s5,-2   # t0=k-2
     bne  t0,0,L3    # branch k!=2
     sub  s0,s1,s2 #k==2 so f=g-h
     jal zero,Exit          # end of case so Exit
L3:  addi t0,s5,-3   # t0=k-3
     bne  t0,0,Exit # branch k!=3
     sub  s0,s3,s4 #k==3 so f=i-j
Exit:
```

```
if(k==0) f=i+j;
  else if(k==1) f=g+h;
    else if(k==2) f=g-h;
      else if(k==3) f=i-j;
```

```
f:s0, g:s1,
h:s2,i:s3, j:s4,
k:s5
```