10/1: Lecture Topics

- Conditional branch instructions
  - slides from 9/29 set
- Unconditional jump instructions
- Instruction encoding

Unconditional Jump

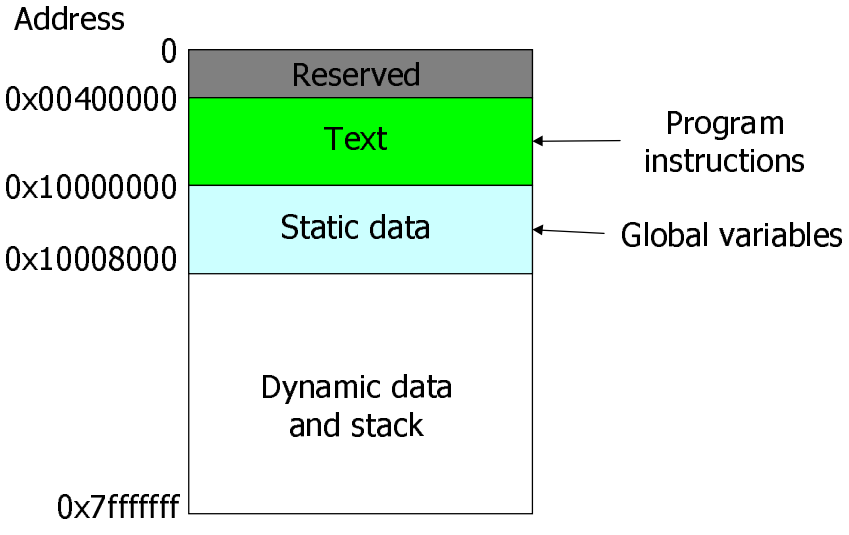
- (Mostly) the same as branch
- No choice about it; just go to the label

- How are branch and jump different?
  - we'll see when we get to instruction encoding

Stored Program

- So far, we've seen that data comes from memory
- It turns out that the program comes from memory also
- Each instruction in the program has an address
- Von Neumann computer (p. 33)

Program Layout (p. 160)

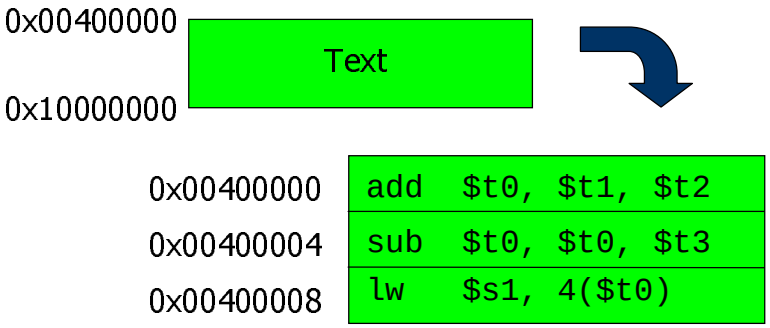


Quick Review of Hexadecimal

- Humans normally use base 10 numbers
- Hexadecimal is base 16
  - use 0-9 as normal, then continue on with a, b, c, d, e, f, then:
  - 10, 11, ... fd, fe, ff, 100, 101, ... ffe, fff
- Hex is convenient for 32-bit addresses
  - Each hex digit comprises 4 binary digits
  - A 32 bit address takes 8 hex digits
- Example: binary 10110011 = 0xb3

The Text Segment

- Let's zoom in on the text segment:



- What's the fib on this slide?

Jump Register

- Now we're ready for the `jr` instruction
- Syntax:

```
jr   $t0
```

- Effect: jump to the instruction *at the address* in $t0

Another Look at Labels

- Labels are text tags

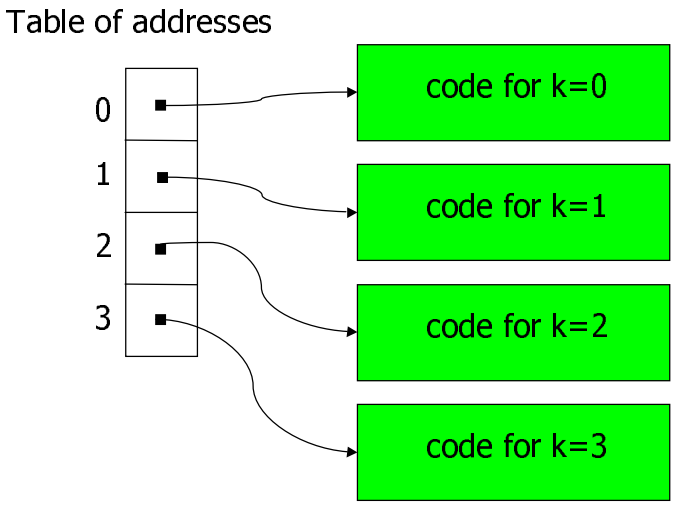
```
loop:  add $t0, $t0, $t0
       bne $t0, $t1, loop
       sw  $t2, 0($t3)
```

- The assembler translates them into addresses and does search-and-replace

C Switch (p. 129)

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

Idea for implementing switch



Implementing C switch

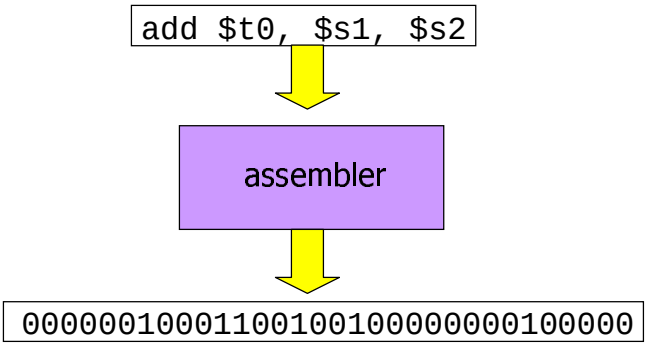
$s0	f
$s1	g
$s2	h
$s3	i
$s4	j
$s5	k
$t4	table addr.

offset	contents
0	addr. L0
4	addr. L1
8	addr. L2
12	addr. L3

```
    add  $t1, $s5, $s5
    add  $t1, $t1, $t1
    add  $t1, $t1, $t4
    lw   $t0, 0($t1)
    jr   $t0
L0: add  $s0, $s3, $s4
    j Exit
L1: add  $s0, $s1, $s2
    j Exit
L2: sub  $s0, $s1, $s2
    j Exit
L3: sub  $s0, $s3, $s4
Exit:
```

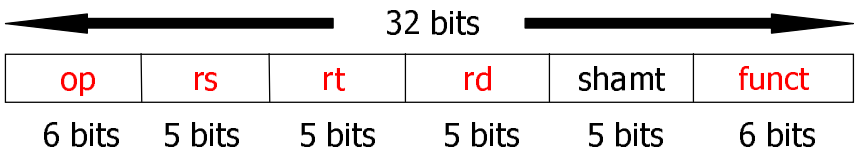
Instruction Encoding

- How does the assembler translate each instruction into bits?



Fields

- Split the 32-bit instruction into *fields:*



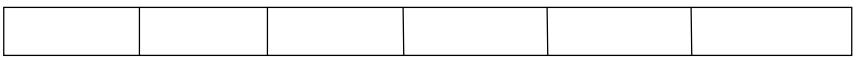
- op+funct: Which instruction?
- rs, rt: Register source operands
- rd: Register destination operand

Encoding an add instruction

- Let's translate:

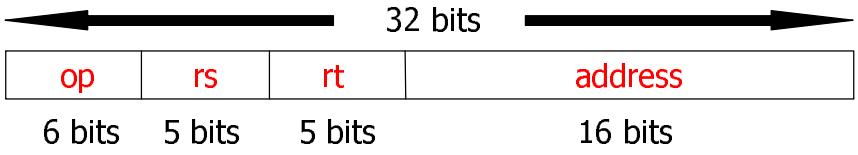
```
add $t0, $s1, $s2
```

- Opcode for `add` is `000000` (p. A-55)
- The funct for `add` is `100000`
- The code for `$t0` is `01000` (p. A-23)
- The code for `$s1` is `10001`
- The code for `$s2` is `10010`



Instruction Formats

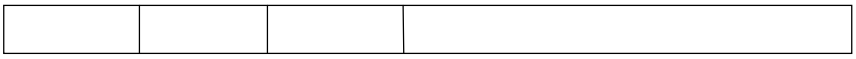
- The 32 bits can be split up more than one way
- For add, it was 6-5-5-5-5-6
  - This is called R-format
- For lw, we use 6-5-5-16 instead
  - This is called I-format



Translating into I-format

```
lw $t0, 100($t1)
```

- The opcode for `lw` is `100011`
- The code for `$t0` is `01000`
- The code for `$t1` is `01001`
- Binary for 100 is



Branch vs. Jump

- Branch instructions use the I-format
  - The destination address is 16 bits
  - You can only branch ±32KB away
- Jump instructions use another format called the J-format
  - The destination address is 26 bits
  - You can jump up to ±32MB away
- What's the tradeoff?

What to Take Away

- I don't care if you:
  - Know how many bits are in each field
  - Know what the opcodes are
  - Know what the codes for the registers are
- I do care that you:
  - Know what fields are
  - Know why we need more than one instruction format