Assembling and Disassembling

Class 31

A Single Byte

- we have a 32-bit word

  `xxxxxxxx 10101110 xxxxxxxx xxxxxxxx`

- we want just the 3rd byte
  1. shift the word 16 places to the right

     `00000000 00000000 xxxxxxxx 10101110`

  2. and the result with the mask 0xFF

     ```
     00000000 00000000 xxxxxxxx 10101110
     00000000 00000000 00000000 11111111
     -----------------------------------
     00000000 00000000 00000000 10101110
     ```

Shift

- the primary MIPS instructions for shift are
  - `sll` shift left logical
  - `srl` shift right logical
- whichever end they shift towards, the other end is filled in with zeros
- these instructions are identical to the C statements

  ```
  dest = source << sh_amt;
  ```

  and

  ```
  dest = source >> sh_amt;
  ```

  respectively
- note that `sll` is equivalent to muliplying by powers of 2, and `srl` is equivalent to dividing by powers of 2

Shift Format

- from the green card, the `srl` instruction is:

  `R  R[rd] = R[rt] >>> shamt  0/02`

  (in C, it's >>, but the green card has the Java >>> unsigned shift right instead)
- this means it uses the R instruction format:

R	opcode	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- the 0/02 means the opcode is 000000 and the function field is 000010
- rs is not used in the instruction, so the rs field is 00000

Shift Assembly

- assume, from the first slide, that the shift amount is 16_{10}, the source value is in $s4 (register 20), and the destination register is $t2 (register 10)
- then we have:

```
srl   $t2, $s4, 16
```

- when assembled, the fields are:

000000	00000	10100	01010	10000	000010
opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

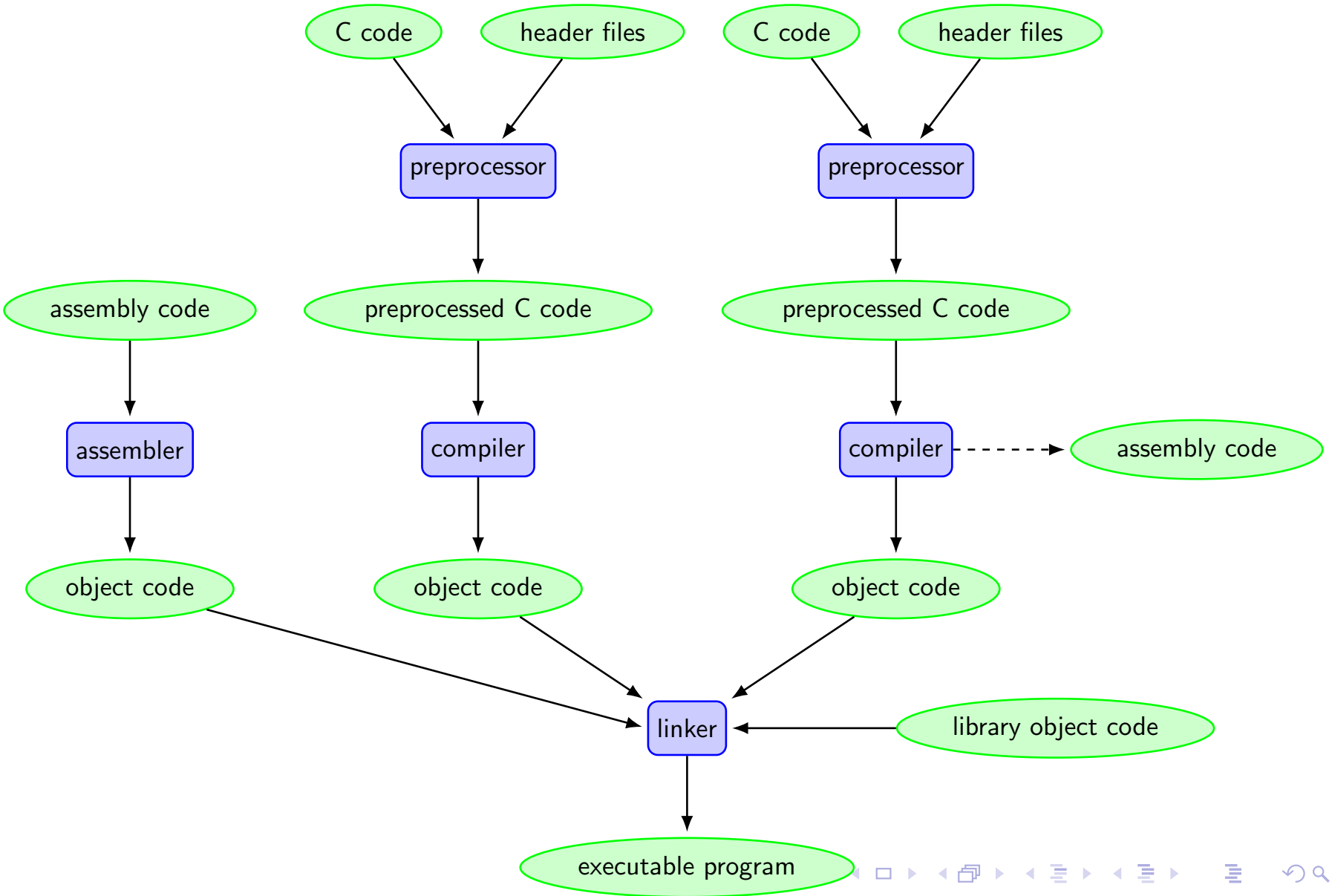
- and so the final machine instruction is:

```
00000000 00010100 01010100 00000010
```

or in hex:

```
0x00 14 54 02
```

C Compilation Process



And

- the second instruction with which we began was bitwise logical `and`

```
and   R   R[rd] = R[rs] & R[rt]    0/24hex
```

- it does not use `shamt`, and so that field is 0

And Immediate

- a similar instruction is and immediate `andi`

```
andi  I  R[rd] = R[rs] & ZeroExtImm
```

- this allows a register value to be and-ed with a literal value
- however, only 16 bits are available for the literal, with the upper 16 bits zero
- so not all literal values are possible
- for example, the following is impossible, and trying to specify this immediate value in assembly language would give an error:

```
andi  $s3, 0xff00
```

```
register: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
mask:     11111111 11111111 00000000 00000000
          -----------------------------------
```

Not

- logical or and logical or immediate, or and ori, are identical to the ands
- the third big logical operator is not
- in their wisdom, the designers of MIPS did not include not
- instead, they provided logical nor

- this is in fact more useful than not, because it can kill two birds with one stone
- it works as binary logical nor with three normal registers
- it works as unary logical not with two registers and zero
- remember from 191:

$$a \downarrow 0 = \neg(a \vee 0) = \neg a$$

```
nor   $s2, $t2, $zero # t2 nor 0 = not(t0)
```

Disassembling

- almost always, we go in the forward direction:

  C code $\rightarrow$ compile $\rightarrow$ execute

  or

  assembly language $\rightarrow$ assemble $\rightarrow$ execute

- but sometimes, we have to go the other direction
- in a debugger, you encounter the machine language instruction:

  `0x8fa40000`

  and you need to know what instruction it is

Disassembling

`0x8fa40000`

- every MIPS instruction has a 6-bit opcode
- all R format instructions have opcode 0, and the instruction is determined by the function field
- so to determine which instruction this is, we must look at the first 6 bits
- the first 8 bits are `0x8f = 1000 1111`
- so the first 6 bits are `10 0011 = 0x23`
- what instruction is this?

Disassembling

`0x8fa40000`

- every MIPS instruction has a 6-bit opcode
- all R format instructions have opcode 0, and the instruction is determined by the function field
- so to determine which instruction this is, we must look at the first 6 bits
- the first 8 bits are `0x8f = 1000 1111`
- so the first 6 bits are `10 0011 = 0x23`
- what instruction is this?
- load word

Disassembling

`0x8fa40000`

- since it's load word, we know the instruction format is I
- we can divide the bits into the proper fields

I	opcode 6 bits	rs 5 bits	rt 5 bits	immediate 16 bits

`100011 11101 00100 0000000000000000`

- we have already determined the opcode is for `lw`
- `rs` is the base register `1 1101 = 0x1d = 29`
  register 29 is `$sp`
- `rt` is the destination register `0 0100 = 0x4 = 4`
  register 4 is `$a0`
- the offset is all zeroes
- together: `lw $a0, 0($sp)`

Making Decisions

- MIPS has just two instructions for conditional branch statements:
  - `beq` branch if equal
  - `bne` branch if not equal
- all ifs, switches, all of the looping constructs, have only these two instructions as building blocks
- this is in contrast to many ISAs which have much more robust sets of branching instructions
- even the tiny Arduino microcontroller has fourteen different conditional branching instructions!
- so, how do we accomplish all the decision logic with just two branch instructions?

Branching

- the two branching instructions in assembly language look like this:

```
beq  register1, register2, label
```

  and

```
bne  register1, register2, label
```

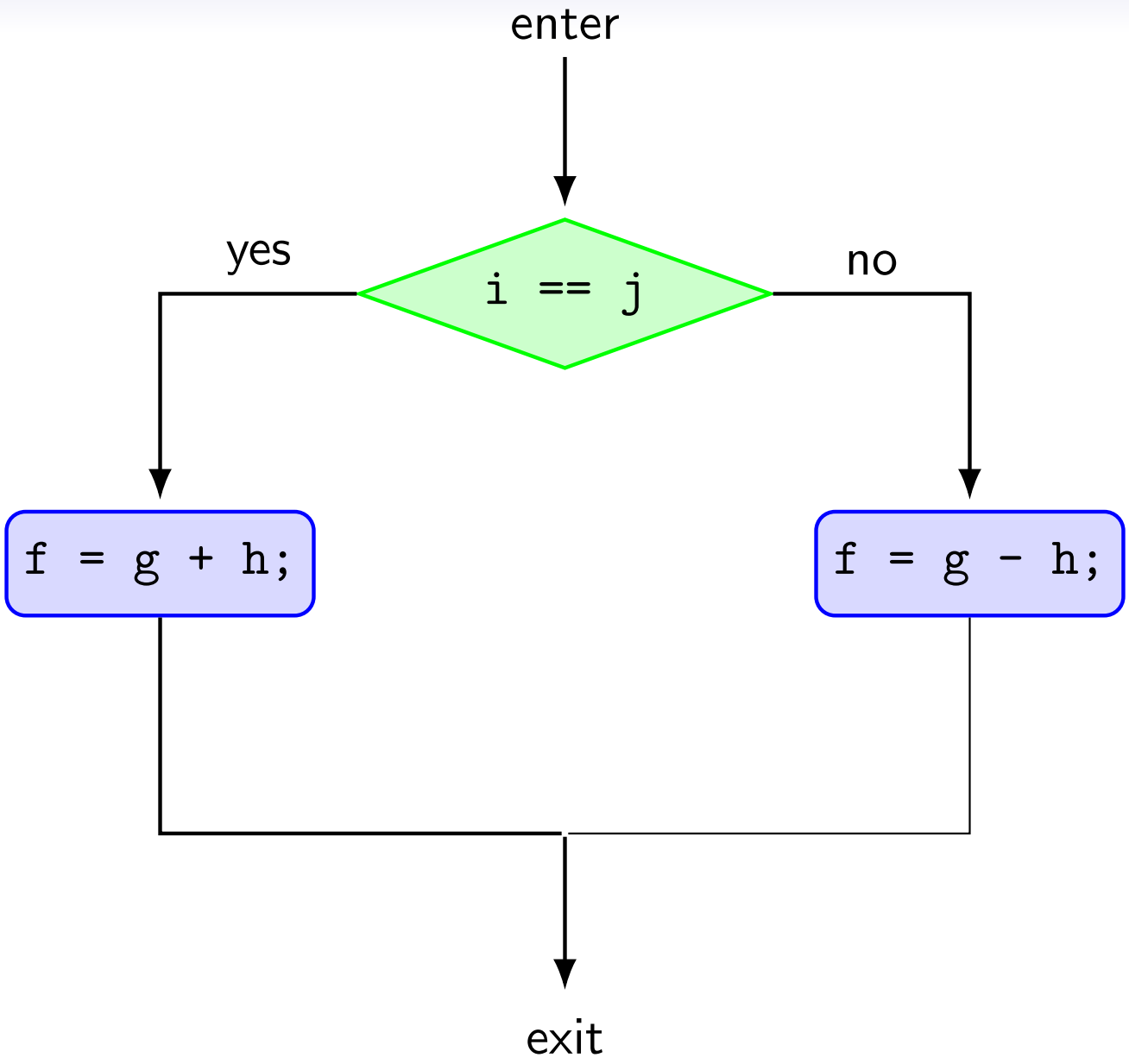
- each means to jump directly to the instruction labeled "label" if the two registers are bit-by-bit identical, or are not identical, respectively
- since these instructions are based on a condition, they are called conditional branches
- in contrast, the three jump instructions are unconditional branches, because they go to an instruction at a different location unconditionally

If Statement

- consider the C if statement:

```
1  if (i == j)
2  {
3    f = g + h;
4  }
5  else
6  {
7    f = g - h;
8  }
```

- the flowchart is shown in the following slide


enter
yes
i == j
no
f = g + h;
f = g - h;
exit

If Statement

- the if-else statement can be implemented with either the `beq` or the `bne`
- it's traditional to branch on the opposite condition than is expressed in the high-level code
- it makes the assembly code mirror the high-level code much more closely
- much less likely to get confused when learning

If Statement

```
1  if (i == j)
2  {
3    f = g + h;
4  }
5  else
6  {
7    f = g - h;
8  }
```

```
1  enter:
2       bne $s3, $s4, else   # go to else if i != j
3       add $s0, $s1, $s2    # f = g + h
4       j   exit             # goto exit
5  else:
6       sub $s0, $s1, $s2    # f = g - h; fall thru
7  exit:
```

While Loop

```
1  while (A[i] == k)
2  {
3    i++;
4  }
```

- assume $s3 is i, $s5 is k, and $s6 is the starting address of A

```
1  loop:
2     sll   $t1, $s3, 2     # $t1 = i * 4
3     addu  $t1, $t1, $s6   # $t1 = &A[i]
4     lw    $t0, 0($t1)     # $t0 = A[i]
5     bne   $t0, $s5, exit  # goto exit if A[i] != k
6     addiu $s3, $s3, 1     # i++
7     j     loop            # goto loop
8  exit:
```

Set

- MIPS has only two conditional branching instructions
- but it also has several instructions that set a register to either 1 or 0 depending on a situation
- these can be used in conjunction with beq and bne

```
slt  $t0, $s3, $s4  # t0 = s3 < s4 ? 1 : 0
slti $t0, $s3, 10   # t0 = s3 < 10 ? 1 : 0
```

Set and Branch

```
1  if (i < j)
2  {
3    f = g + h;
4  }
5  else
6  {
7    f = g - h;
8  }
```

```
1  enter:
2       slt $t0, $s3, $s4      # t0 = s3 < s4 ? 1 : 0
3       beq $t0, $zero, else   # goto else if t0 is 0
4       add $s0, $s1, $s2      # f = g + h
5       j   exit               # goto exit
6  else:
7       sub $s0, $s1, $s2      # f = g - h; fall thru
8  exit:
```

Signed vs. Unsigned

- suppose our registers were 4 bits instead of 32
- let \$s0 = 1001 and \$s1 = 0001
- what is the result of the following?

```
slt $t0, $s0, $s1
```

Signed vs. Unsigned

- suppose our registers were 4 bits instead of 32
- let \$s0 = 1001 and \$s1 = 0001
- what is the result of the following?

```
slt $t0, $s0, $s1
```

- the value in \$s0 is -7 if \$s0 is signed
- the value in \$s0 is 9 if \$s0 is unsigned
- the value in \$s1 is 1 regardless of signed or unsigned

- slt is a signed less-than
- if we want unsigned, we use sltu instead

Signed vs. Unsigned

- assembly language has no data types
- all registers are simply bits

- but instructions make huge assumptions about data types
- if you the programmer know that a register should be interpreted as an unsigned integer, you must use sltu
- and if signed, use slt
- using the wrong instruction will give the wrong answer