Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 4
Decisions

FIT 1008
Introduction to Computer Science

What we have seen

- The MIPS R2000 architecture
  - 32 general purpose registers
  - Special purpose registers (HI, LO, PC, IR, etc)
  - ALU
  - Memory segments (text, data, heap, stack)
- The fetch-decode-execute cycle
- The assembly language and assembler directives
- MIPS instruction set

Objectives for this lecture

- To put the MIPS branch and jump instructions into context
- To understand how they are used to translate selection (if-else)
- To understand how they are used to translate iteration (loops)
  - while
  - for
- To see the MIPS instruction format

Blast from the past: the **goto** statement

- A **label** is an identifier for a program position (i.e., for a line of code)

- The **goto** statement performs an unconditional jump to its label argument

- It promotes code whose control flow is **extremely difficult to understand**

- That is why it is not supported by many languages, including Python

- However, in MIPS the equivalent **jump** instruction is all we've got!

If Python had a `goto` statement …

```
# Code could be this ugly!?

      print(1)
      goto apple

   orange:
      print(3)
      goto pomegranate

   apple:
      print(2)
      goto orange

   pomegranate:
      print(4)
```

Jump Instructions

- jump (go) to label

```
j foo              # set PC = foo
                   # so, go to foo
```

- jump to label and link (remember origin)

```
jal foo            # $ra = PC+4; PC = foo, so same
                   # but setting a return address
```

- jump to address contained in register

```
jr $t0             # set PC = $t0, so go to the
                   # address contained in $t0
```

- jump to register and link (remember origin)

```
jalr $t0           # $ra = PC+4; PC = $t0, same
                   # but setting a return address
```

MIPS **jump** instruction

```
                # print number 1
                j   apple
orange:         #print number 3
                j   pomegranate
apple:          #print number 2
                j   orange
pomegranate:
                # print number 4
                # exit system call
```

Selection

- **Selection** is how programs **make choices**

- In Python, with **if**, **if-else**, **if-elif-else** (like switch cases)

- Achieved by selectively not executing some lines of code

Comparison Instructions

- set less than

```
slt $t0,$t1,$t2      # if $t1<$t2 then $t0=1
                     # else $t0 = 0
```

- set less than immediate

```
slti $t0,$t1,1       # if $t1<1 then $t0=1
                     # else $t0 = 0
```

- Note: comparisons are performed by the ALU, so comparison instructions are really arithmetic ones

Conditional Branch Instructions

- branch if equal to

```
beq $t1,$t2,foo      # if $t1==$t2 goto foo
```

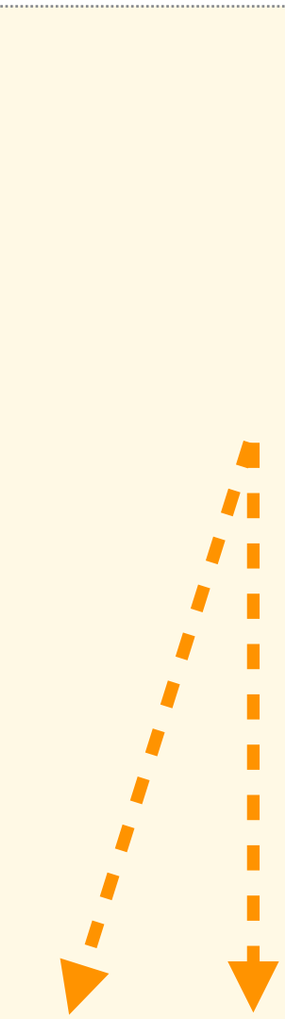
- branch if not equal to

```
bne $t1,$t2,foo      # if $t1!=$t2 goto foo
```

negative.py

```
n = int(input("Enter int: "))

if n < 0:
        print("Negative")
```

If n ≥ 0 goto exit



```
        lw    $t0, n        # if n >= 0 goto exit
        slt   $t1, $t0, $0
        beq   $t1, $0, exit
        la    $a0, negative     # print negative
        addi  $v0, $0, 4
        syscall
exit:   addi  $v0, $0, 10       # exit program
        syscall
```

negative.asm

```
                .data
prompt:     .asciiz   "Enter int: "
negative:   .asciiz   "Negative"
n:          .word      0

                .text
        la      $a0, prompt   # print prompt
        add     $v0, $0, 4
        syscall

        addi  $v0, $0, 5      # read n
        syscall
        sw      $v0, n
```

if – else statement

```
# compute n % 2
lw     $t0, n
addi   $t1, $0, 2
# $t0 = n % 2
div    $t0, $t1
mfhi   $t0
# if $t0 <> 0
goto else
bne   $t0, $0, else
```

```
    # print n
    # print even
    j exit
else:
        # print n
        # print odd
exit:
        # exit
```

Reminder: Iteration

- Iteration is the repetition of a section of code
  - In Python, with while, for
    - while tests condition before loop entry
    - for is a shorthand for while

- Achieved by sending control from the end of the loop back to the beginning
  - Test some condition to prevent infinite loop

factorial.py

```
f = 1

n = int(input("Enter int: "))

while n > 0:

    f = f * n

    n -= 1

print(f)
```

factorial.asm

```
                # set up strings
                # set up n = 0 and f = 1
                # read n
loop:
                # if n <= 0 goto endloop
                # f = f * n
                # n -= 1
                # goto loop
endloop:
                # print f
                # exit
```

Let's look at this

if n <= 0 goto endloop

```
    lw    $t0, n

    slt   $t1, $0, $t0

    beq   $t1, $0, endloop
```

setup

```
        .data
prompt: .asciiz "Enter int: "
     f: .word   1
     n: .word   0
```

```
        .text
# print prompt
  la       $a0, prompt
  addi     $v0, $0, 4
  syscall
# read n
  addi     $v0, $0, 5
  syscall
  sw       $v0, n
```

loop

```
# if n <= 0 goto endloop

loop:

    lw   $t0, n

    slt  $t1, $0, $t0

    beq  $t1, $0, endloop

    lw   $t1, f

    # f = f * n

    mult $t1, $t0

    mflo $t1

    sw   $t1, f

    # n = n - 1

    lw   $t0, n

    addi $t0, $t0, -1

    sw   $t0, n

    j    loop

endloop:
```

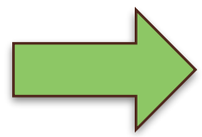
endloop

```
endloop:

    lw    $a0, f          # print f

    addi  $v0, $0, 1

    syscall

    addi  $v0, $0, 10     # exit

    syscall
```

Iteration: for

- A **`for`** loop is essentially a simpler version of a **`while`** loop:
  - Initialisation, condition and increment code all in one place
- To translate a **`for`** loop into MIPS, write it as a **`while`** loop

```
for i in range(init, cond, inc):
        body
```



```
i = init
while (cond):
    body
    inc
```

MIPS Instruction Format

- Remember: every MIPS instruction is 32-bits in size and occupies 4 bytes of memory
- Remember: each instruction contains
  - opcode
    - operation code: specifies type of instruction
  - operands
    - values or location to perform operation on
      - registers
      - immediate (constant) numbers
      - labels (addresses of other lines of program)

MIPS Instruction Format

R (for "register") format instruction: three registers

```
sub $t0, $t1, $t2
```

subtract the contents of register $t2 from the contents of register $t1; put the result in register $t0

I (for "immediate") format instruction: two registers and one immediate operand

```
addi $v0, $a2, 742
```

add the immediate number 742 with the contents of register $a2; put the result in register $v0

J (for "jump") format instruction: has a line label (an address) as its only operand

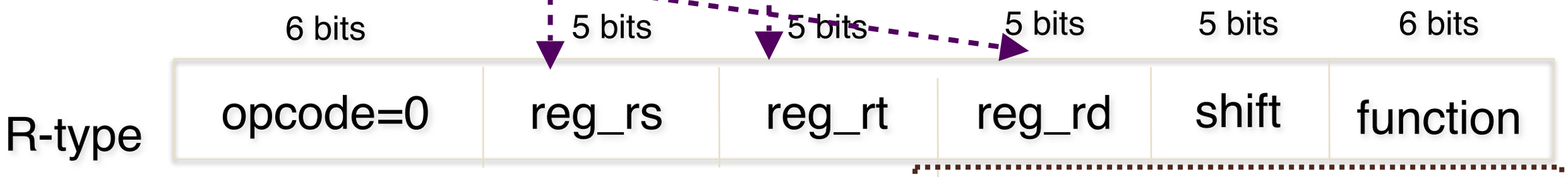
```
j foo
```

jump (go) to the line with the label foo and continue running from there

MIPS Instruction Format

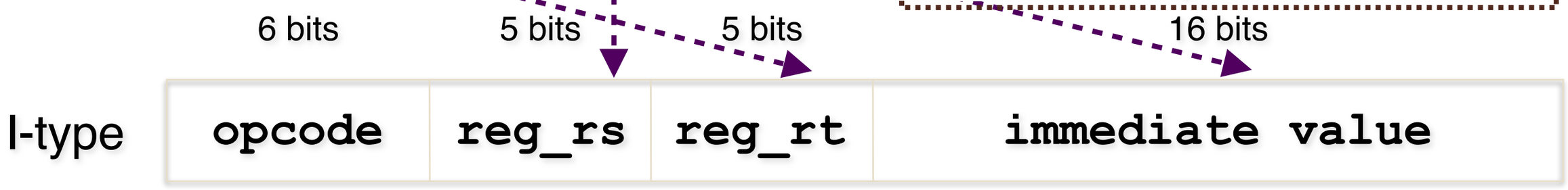
R (for "register") format instruction: three registers

```
sub $t0, $t1, $t2
```



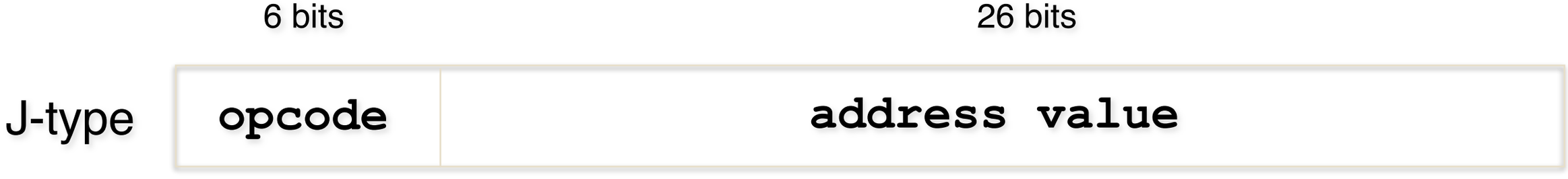
I (for "immediate") format instruction: two registers and one immediate operand

```
addi $v0, $a2, 742
```



J (for "jump") format instruction: has a line label (an address) as its only operand

```
j foo
```



I-type Instruction: Example

opcode determines how remaining bits are to be interpreted as operands

Instruction's components encoded in binary

opcode (6 bits): 001000_2
(8_{10}) means "add immediate"

Summary

- MIPS branch and jump instructions
- Selection
  - if-else
- Iteration (loops)
  - while
  - for
- Instruction Format
  - R type
  - I type
  - J type