EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 7_3: MIPS ISA

Xiaolin Xu
Department of ECE
Northeastern University

Generating Constants

- 16-bit constants using `addi`:
- Recall why we store all-0 in $0!

C Code

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

Generating Constants

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Generating Constants

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Why not

```
addi $s0, $0, 0FEDC8765
```

Generating Constants

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Why not

```
addi $s0, $0, 0FEDC8765
```

Why not "load immediate"

Generating Constants

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Why not

```
addi $s0, $0, 0FEDC8765
```

Why not "load immediate" —> it will be from memory, not register anymore!

Generating Constants

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Why not

```
addi $s0, $0, 0FEDC8765
```

Why not "load immediate" —> it will be from memory, not register anymore!

—> li is a pseudo instruction, MIPS does NOT have a real li instruction

Multiplication and Division Instructions

- Special!
- **Multiplying two 32-bit numbers —> 64-bit**
- **Dividing two 32-bit numbers —> 32-bit quotient and 32-bit remainder**

Multiplication

- Special registers: `lo`, `hi`
- 32 × 32 multiplication, 64 bit result
  - `mult $s0, $s1`
  - Result in {`hi`, `lo`}
  - `hi = 32-bit MSB`, `lo = 32-bit LSB`

Division

- 32-bit division, 32-bit quotient, remainder
  - `div $s0, $s1`
  - Quotient in `lo`
  - Remainder in `hi`

Division

- Moves from `lo/hi` special registers
  - `mflo $s2`
  - `mfhi $s3`

`Where are the source registers?`

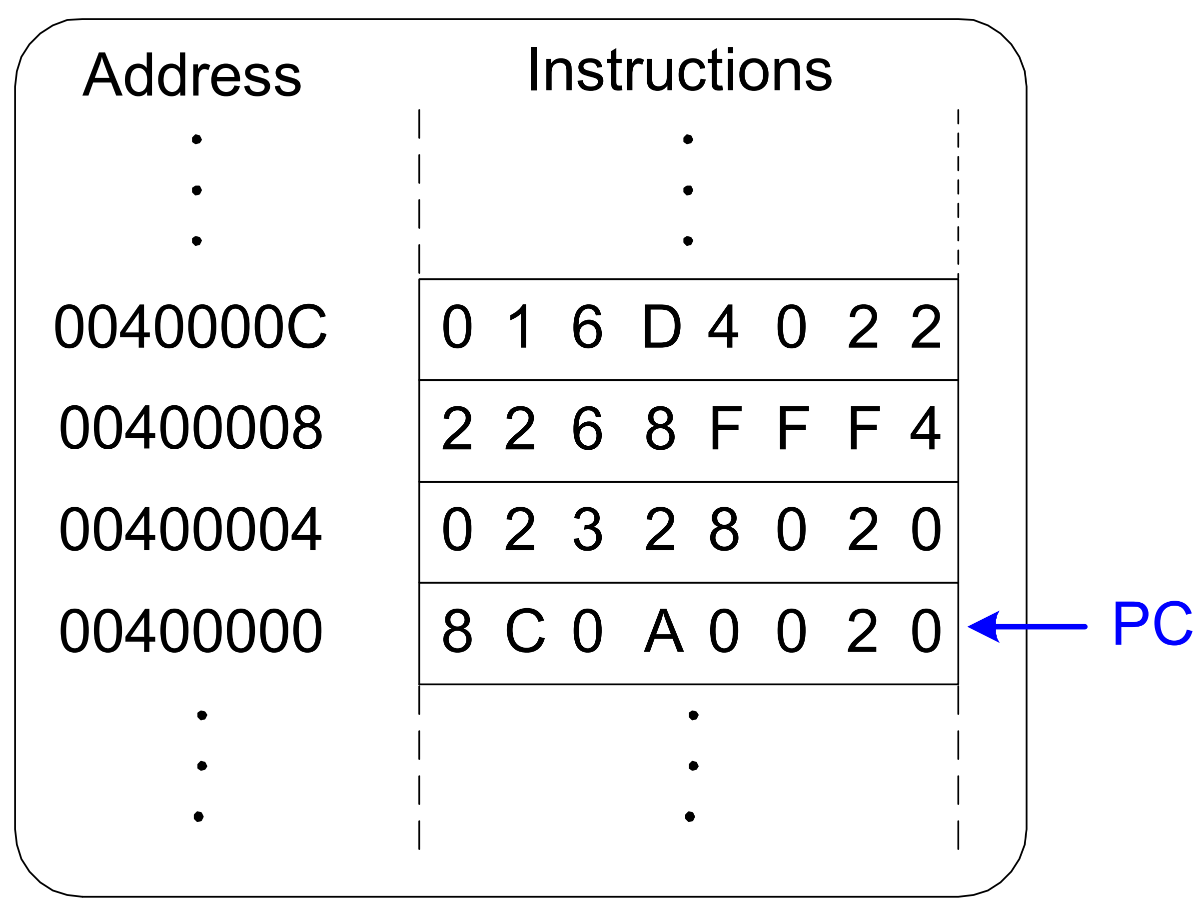
Branching

- A computer is not just simply calculating, but **a decision maker!**
- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
  - **Unconditional**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)

Review: The Stored Program w/o Branch

Assembly Code	Machine Code
`lw   $t2, 32($0)`	`0x8C0A0020`
`add  $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub  $t0, $t3, $t5`	`0x016D4022`

Stored Program



Main Memory

Conditional Branching (beq)

 # MIPS assembly **Results per line?**

```
addi $s0, $0, 4
addi $s1, $0, 1
sll  $s1, $s1, 2
beq  $s0, $s1, target
addi $s1, $s1, 1
sub  $s1, $s1, $s0
```

Is this "targeted" instruction executed or not? and its results?

```
target:        # label
 add  $s1, $s1, $s0
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

Conditional Branching (beq)

 # MIPS assembly **Results per line?**

```
addi $s0, $0, 4        # $s0 = 0 + 4 = 4
addi $s1, $0, 1        # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2       # $s1 = 1 << 2 = 4
beq  $s0, $s1, target  # branch is taken
addi $s1, $s1, 1       # not executed
sub  $s1, $s1, $s0     # not executed
```

Is this "targeted" instruction executed or not? and its results?

```
target:          # label
 add  $s1, $s1, $s0
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

Conditional Branching (beq)

```
# MIPS assembly                Results per line?
 addi $s0, $0, 4              # $s0 = 0 + 4 = 4
 addi $s1, $0, 1              # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2             # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target        # branch is taken
 addi $s1, $s1, 1             # not executed
 sub  $s1, $s1, $s0           # not executed
```

Is this "targeted" instruction executed or not? and its results?

```
target:          # label
 add  $s1, $s1, $s0           # $s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

The Branch Not Taken (bne)

```
# MIPS assembly
  addi   $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi   $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll    $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne    $s0, $s1, target
  addi   $s1, $s1, 1
  sub    $s1, $s1, $s0



target:
  add    $s1, $s1, $s0
```

The Branch Not Taken (bne)

 # MIPS assembly

```
  addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne   $s0, $s1, target
  addi  $s1, $s1, 1
  sub   $s1, $s1, $s0
```

Is this "targeted" instruction executed or not? and its results?

```
target:
  add   $s1, $s1, $s0
```

The Branch Not Taken (bne)

 # MIPS assembly

```
addi   $s0, $0, 4          # $s0 = 0 + 4 = 4
addi   $s1, $0, 1          # $s1 = 0 + 1 = 1
sll    $s1, $s1, 2         # $s1 = 1 << 2 = 4
bne    $s0, $s1, target    # branch not taken
addi   $s1, $s1, 1         # $s1 = 4 + 1 = 5
sub    $s1, $s1, $s0       # $s1 = 5 - 4 = 1
```

Is this "targeted" instruction executed or not? and its results?

```
target:
  add    $s1, $s1, $s0
```

The Branch Not Taken (bne)

```
# MIPS assembly
  addi  $s0, $0, 4         # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1         # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2        # $s1 = 1 << 2 = 4
  bne   $s0, $s1, target   # branch not taken
  addi  $s1, $s1, 1        # $s1 = 4 + 1 = 5
  sub   $s1, $s1, $s0      # $s1 = 5 – 4 = 1
```

Is this "targeted" instruction executed or not? and its results?

```
target:
  add   $s1, $s1, $s0      # $s1 = 1 + 4 = 5
```

Unconditional Branching: jump (j)

```
# MIPS assembly
addi $s0, $0, 4          # $s0 = 4
addi $s1, $0, 1          # $s1 = 1
j    target              # jump to target
sra  $s1, $s1, 2
addi $s1, $s1, 1
sub  $s1, $s1, $s0

target:
add  $s1, $s1, $s0
```

Unconditional Branching: jump (j)

```
# MIPS assembly
  addi $s0, $0, 4        # $s0 = 4
  addi $s1, $0, 1        # $s1 = 1
  j    target            # jump to target
  sra  $s1, $s1, 2       # not executed
  addi $s1, $s1, 1       # not executed
  sub  $s1, $s1, $s0     # not executed

target:
  add  $s1, $s1, $s0     # $s1 = 1 + 4 = 5
```

Unconditional Branching: jump register (jr)

```
# MIPS assembly
0x00002000        addi $s0, $0, 0x2010
0x00002004        jr   $s0
0x00002008        addi $s1, $0, 1
0x0000200C        sra  $s1, $s1, 2
0x00002010        lw   $s3, 44($s1)
```

`jr` is an **R-type** instruction.

Unconditional Branching: jump register (jr)

```
# MIPS assembly
0x00002000     addi $s0, $0, 0x2010
0x00002004     jr   $s0              # $s0 = 0x00002010
0x00002008     addi $s1, $0, 1       # not executed
0x0000200C     sra  $s1, $s1, 2      # not executed
0x00002010     lw   $s3, 44($s1)
```

`jr` is an **R-type** instruction.

High-Level Code Constructs

- if statements
- if/else statements
- while loops
- for loops

If Statement

- Fill the assembly code (bne, label, etc)

C Code

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

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

____

____

____

If Statement

C Code

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

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

If / Else Statement

- Fill the assembly code (bne, label, etc)

C Code

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

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

____

____

____

____

If / Else Statement

C Code

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

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
        bne $s3, $s4, L1
        add $s0, $s1, $s2
        j   done
L1:     sub $s0, $s0, $s3
done:
```

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x

      addi $s0, $0, 1    # pow = 1
_______________________
                         # x = 0
_______________________
                         # t0 = 128 for
_______________________    comparison
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x

       addi $s0, $0, 1
       add  $s1, $0, $0
       addi $t0, $0, 128
while:                        # if pow=128,
                                exit
       ______________________
                              # pow = 2* pow
       ______________________
                              # x = x + 1
       ______________________
done:
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x

       addi $s0, $0, 1
       add  $s1, $0, $0
       addi $t0, $0, 128
while: beq  $s0, $t0, done
       sll  $s0, $s0, 1
       addi $s1, $s1, 1
       j    while
done:
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

For Loops

```
for (initialization; condition; loop operation)
  statement
```

- `initialization:` executes before the loop begins
- `condition:` is tested at the beginning of each iteration
- `loop operation:` executes at the end of each iteration
- `statement:` executes each time the condition is met

For Loops

High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
```

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
       addi $s1, $0, 0
       add  $s0, $0, $0
       addi $t0, $0, 10
for:   beq  $s0, $t0, done
       add  $s1, $s1, $s0
       addi $s0, $s0, 1
       j    for
done:
```