

# Lecture 5: More Instructions, Procedure Calls

---

- Today's topics:
  - Memory layout, numbers, control instructions
  - Procedure calls
  - Recaps: cycle time and frequency, addr/value

# Example

---

Convert to assembly:

C code: `d[3] = d[2] + a;`

Assembly (same assumptions as previous example):

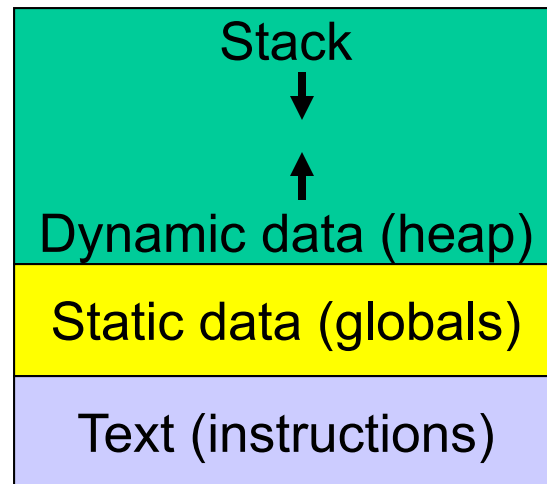
```
lw    $s0, 0($gp)    # a is brought into $s0
lw    $s1, 20($gp)   # d[2] is brought into $s1
add   $t1, $s0, $s1  # the sum is in $t1
sw    $t1, 24($gp)   # $t1 is stored into d[3]
```

Assembly version of the code continues to expand!

# Memory Organization

---

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to \$fp as \$sp may change during the execution of the procedure
- \$gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap



# Recap – Numeric Representations

---

- Decimal  $35_{10} = 3 \times 10^1 + 5 \times 10^0$
- Binary  $00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0$
- Hexadecimal (compact representation)  
 $0x23$  or  $23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0$

0-15 (decimal)  $\rightarrow$  0-9, a-f (hex)

| Dec | Binary | Hex | Dec | Binary | Hex | Dec | Binary | Hex | Dec | Binary | Hex |
|-----|--------|-----|-----|--------|-----|-----|--------|-----|-----|--------|-----|
| 0   | 0000   | 00  | 4   | 0100   | 04  | 8   | 1000   | 08  | 12  | 1100   | 0c  |
| 1   | 0001   | 01  | 5   | 0101   | 05  | 9   | 1001   | 09  | 13  | 1101   | 0d  |
| 2   | 0010   | 02  | 6   | 0110   | 06  | 10  | 1010   | 0a  | 14  | 1110   | 0e  |
| 3   | 0011   | 03  | 7   | 0111   | 07  | 11  | 1011   | 0b  | 15  | 1111   | 0f  |

# Instruction Formats

---

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

*R-type instruction*                      add    \$t0, \$s1, \$s2

|        |        |        |        |           |          |
|--------|--------|--------|--------|-----------|----------|
| 000000 | 10001  | 10010  | 01000  | 00000     | 100000   |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits    | 6 bits   |
| op     | rs     | rt     | rd     | shamt     | funct    |
| opcode | source | source | dest   | shift amt | function |

*I-type instruction*                      lw    \$t0, 32(\$s3)

|        |        |        |          |
|--------|--------|--------|----------|
| 6 bits | 5 bits | 5 bits | 16 bits  |
| opcode | rs     | rt     | constant |

# Logical Operations

---

| Logical ops    | C operators | Java operators | MIPS instr |
|----------------|-------------|----------------|------------|
| Shift Left     | <<          | <<             | sll        |
| Shift Right    | >>          | >>>            | srl        |
| Bit-by-bit AND | &           | &              | and, andi  |
| Bit-by-bit OR  |             |                | or, ori    |
| Bit-by-bit NOT | ~           | ~              | nor        |

# Control Instructions

---

- Conditional branch: Jump to instruction L1 if register1 equals register2: `beq register1, register2, L1`  
Similarly, `bne` and `slt` (set-on-less-than)
- Unconditional branch:  
`j L1`  
`jr $s0` (useful for big jumps and procedure returns)

Convert to assembly:

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

# Control Instructions

---

- Conditional branch: Jump to instruction L1 if register1 equals register2: `beq register1, register2, L1`  
Similarly, `bne` and `slt` (set-on-less-than)
- Unconditional branch:  
`j L1`  
`jr $s0` (useful for big jumps and procedure returns)

Convert to assembly:

|             |                                         |
|-------------|-----------------------------------------|
| if (i == j) | <code>bne \$s3, \$s4, Else</code>       |
| f = g+h;    | <code>add \$s0, \$s1, \$s2</code>       |
| else        | <code>j Exit</code>                     |
| f = g-h;    | <code>Else: sub \$s0, \$s1, \$s2</code> |
|             | <code>Exit:</code>                      |



# Example

---

Convert to assembly:

```
while (save[i] == k)
    i += 1;
```

Values of i and k are in \$s3 and  
\$s5 and base of array save[] is  
in \$s6

# Example

Convert to assembly:

```
while (save[i] == k)
    i += 1;
```

Values of i and k are in \$s3  
and \$s5 and base of array  
save[] is in \$s6

```
Loop: sll    $t1, $s3, 2
      add    $t1, $t1, $s6
      lw     $t0, 0($t1)
      bne    $t0, $s5, Exit
      addi   $s3, $s3, 1
      j      Loop
```

Exit:

```
      sll    $t1, $s3, 2
      add    $t1, $t1, $s6
Loop: lw     $t0, 0($t1)
      bne    $t0, $s5, Exit
      addi   $s3, $s3, 1
      addi   $t1, $t1, 4
      j      Loop
```

Exit:

# Registers

---

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : \$zero      always stores the constant 0
  - Regs 2-3 : \$v0, \$v1    return values of a procedure
  - Regs 4-7 : \$a0-\$a3    input arguments to a procedure
  - Regs 8-15 : \$t0-\$t7    temporaries
  - Regs 16-23: \$s0-\$s7    variables
  - Regs 24-25: \$t8-\$t9    more temporaries
  - Reg 28 : \$gp            global pointer
  - Reg 29 : \$sp            stack pointer
  - Reg 30 : \$fp            frame pointer
  - Reg 31 : \$ra            return address

# Procedures

---

- Local variables, AR, \$fp, \$sp
- Scratchpad and saves/restores
- Arguments and returns
- jal and \$ra