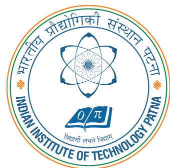
CS321: Computer Architecture

Instruction Set Architecture



Arijit Mondal

Dept. of Computer Science & Engineering

Indian Institute of Technology Patna

arijit@iitp.ac.in

Instructions

- Language for computer hardware
- Different computers may have different instruction sets
- However they are of similar nature
- Basic operations need to be supported
- Early computer had very simple instruction set

Assembly language

- MIPS is one of the most popular languages in academic community
  - Belongs to RISC family
  - ARM ISA has large similarity
- ARM ISA is most poupular in the embedded systems
  - Belongs to RISC family
  - Very popular in 32 bits processor series
- X86 / X86-64 ISA mostly dominates in servers, heavy computation domain
  - Belongs to CISC family
  - Instructions have different size

Arithmetic operation

- `add a, b, c`
  - **Adds the two variables `b` and `c` and stores the result in `a`**
- **Add four variables `b, c, d, e` and store the result in `a`**

```
add a, b, c
add a, a, d
add a, a, e
```

Arithmetic operation (contd.)

- **Two assignments:** `a=b+c; d=a-e;`

```
add a, b, c
sub d, a, e
```

- **Complex assignments:** `f=(g+h)-(i+j)`

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

Design principle - 1

- Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Operand for arithmetic operation

- Number of operand is restricted
- Uses special location ie. *register*
- For MIPS, size of the register is 32 bits
- Three operands of arithmetic instruction must be chosen from the registers

Design principle - 2

- Smaller is faster
  - Large number of registers increases the clock cycle time
  - Hardware cost
  - Trade off between cost and performance
  - To conserve energy

Arithmetic operation using registers (MIPS)

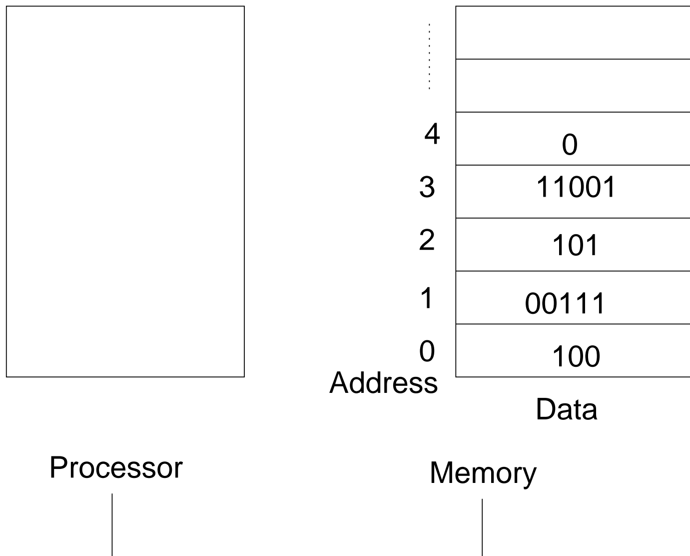
- **Complex assignments:** `f=(g+h)-(i+j)`
  - **Let the variables** `f, g, h, i, j` **are assigned to register** `$s0, $s1, $s2, $s3, $s4`
  - **We need two temporary registers**

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

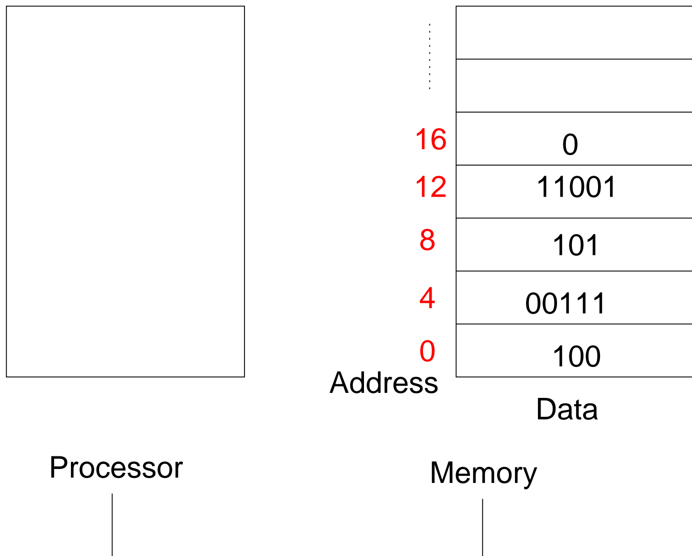
Operands from memory

- Complex data structures like `arrays` and `structures`
- All data may not be available in the registers
- The data are usually stored in memory
- How can a computer represent and access such data?

Memory addresses



Memory addresses for MIPS



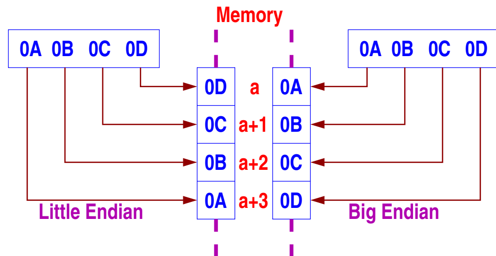
Arithmetic operation using operand from memory

- **Assignments:** `g=h+A[8]`
  - **Let the variables** `g, h` **are assigned to register** `$s1, $s2`
  - **Let** `$s3` **contains** ***base address*** **of array** `A`
  - **Need to use a temporary register** `$t0` **(say) to store the data from memory**

```
lw $t0, 32($s3)
add $s1, $s2, $t0
```

Memory

- Arrays and structures are allocated in memory
- Compiler places proper start address into data transfer function
- In MIPS, memory is 32 bit wide (word)
- Each byte is addressable
- A *little-endian* machine stores the least significant byte first. ARM belongs to this group.
- MIPS belongs to *big-endian* group



Register vs. Memory

- Register are faster to access than memory
- Operating on memory data requires extra load/store call. More instructions get executed.
- Compiler tries to put most frequently data in register and rest are in memory
- To achieve highest performance and conserve energy, compiler must use the register efficiently

Constant or Immediate operand

- Add 4 to register 3
  - Need to load 4 in a register from memory

    ```
    lw $t0, AddrConst4($s1)
    add $s3, $s3, $t0
    ```
- Extra load operation required
- To avoid such load, one of the operands of the instruction be a constant
  - `addi $s3, $s3, 4`

Design principle - 3

- Make the common case fast
  - Constant operands occurs frequently
  - By including constants in arithmetic instruction extra call for load can be avoided
  - This will improve performance both in terms of time and power

Binary numbers

0 0 0 0	0 0 0 0	0 0 0 0	1 1 1 0

15 .... 3 2 1 0

MSB LSB

- $x15 \times 2^{15} + x15 \times 2^{14} + \ldots + x1 \times 2^{1} + x0 \times 2^{0}$
- How to handle signed number?
  - One bit may be reserved for sign, rest of the bits will denote the magnitude
    - Sign location - left or right?
    - How to handle ± 0?
    - How to set sign bit for add operation?
    - Subtraction of a large number from a small one?

2s complement signed number

- Leading 0 means positive and leading 1 means negative number
- For 16 bit representation, 0 to $2^{15} - 1$ will be represented as before
- Bit pattern $1000\ldots000$ will be treated as -2^{15}
- Bit pattern $1111\ldots111$ will be treated as -1
- $x15 \times -2^{15} + x15 \times 2^{14} + \ldots + x1 \times 2^{1} + x0 \times 2^{0}$

Operation on 2s complement numbers

- Let X be binary number represented in 2s complement form. Find $X + \overline{X}$
- Negation of a number?
- Sign extension

Representation of R-Instruction (MIPS)

op	rs	rt	rd	shamt	funct
6bits	5bits	5bits	5bits	5bits	6bits

- `add rs, rt, rd`
- op — Basic operation of instruction
- rs — First register source operand
- rt — Second register source operand
- rd — The register destination operand
- shamt — Shift amount
- funct — Function code, the specific variant of operation in the op field

Register convention (MIPS)

- MIPS has 32 registers
- `$s0 - $s7` – Register number from 16 to 23 (temporary values)
- `$t0 - $t7` – Register number from 8 to 15 (saved values)
- MIPS register 0 (`$zero`) is the constant 0. It cannot be over written
  - Useful for common oeprations
  - Move between registers — `add $t2, $s2, $zero`

Example (MIPS)

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

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

Example (MIPS)

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

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

- `addi $s1, $s2, 4`

op	rs	rt	constant or addr
6bits	5bits	5bit	16bits
8	17	18	4

Data transfer instruction format (MIPS)

- **Instruction format**

op	rs	rt	constant or addr
6 bits	5 bits	5 bits	16 bits

- **Example:** `lw $t0, 32($s3)`

Data transfer instruction format (MIPS)

- **Instruction format**

op	rs	rt	constant or addr
6 bits	5 bits	5 bits	16 bits

- **Example:** `lw $t0, 32($s3)`

op	rs	rt	constant or addr
35	19	8	32

Assembly to binary (MIPS)

- **Assignment operation:** `A[300]=h+A[300]`
- **Base addrress of A is in `$t1` and h is in `$s2`**

```
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

- **Binary code**

<table>
<tr><td rowspan="2">op</td><td rowspan="2">rs</td><td rowspan="2">rt</td><td colspan="3">address or</td></tr>
<tr><td>rd</td><td>shamt</td><td>funct</td></tr>
</table>

Assembly to binary (MIPS)

- **Assignment operation:** `A[300]=h+A[300]`
- **Base addrress of A is in `$t1` and h is in `$s2`**

```
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

- **Binary code**

op	rs	rt	address or		
			rd	shamt	funct
35	9	8	1200		

Assembly to binary (MIPS)

- **Assignment operation:** `A[300]=h+A[300]`
- **Base addrress of A is in `$t1` and h is in `$s2`**

```
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

- **Binary code**

op	rs	rt	address or		
			rd	shamt	funct
35	9	8	1200		
0	18	8	8	0	32

Assembly to binary (MIPS)

- **Assignment operation:** `A[300]=h+A[300]`
- **Base addrress of A is in `$t1` and h is in `$s2`**

```
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

- **Binary code**

op	rs	rt	address or		
			rd	shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Logical operation (MIPS)

- **Shift left — `sll` (<< in C)**
- **Shift right — `srl` (>> in C)**
- **Bit-by-bit AND — `and`, `andi` (& in C)**
- **Bit-by-bit OR — `or`, `ori` (| in C)**
- **Bit-by-bit NOT — `nor` (~ in C)**

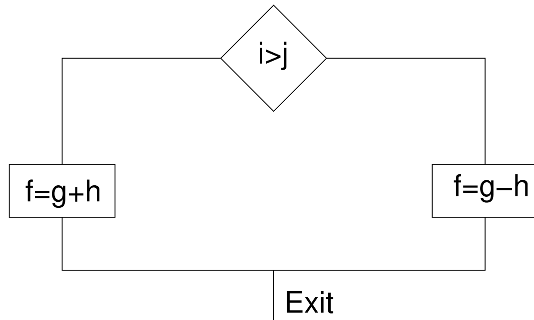
Logical operation example (MIPS)

- **Shift left —** `sll $t2, $s0, 4`

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

- **AND —** `and $t0, $t1, $t2`
- **OR —** `or $t0, $t1, $t2`
- **NOT — It is done using** `nor` **instruction**
  - `nor $t0, $t1, $t3`
  - **Assuming** `$t3` **has the value 0**
  - `nor $t0, $t1, $zero`
- `and`, `or` **instructions have** `andi`, `ori`
- `nor` **does not have any immediate version**

Instructions for making decision (MIPS)




```
beq register1, register2, L1
bne register1, register2, L1
```

If-then-else (MIPS)

- `if(i==j) f=g+h; else f=g-h;`
- `f, g, h, i, j` are in `$s0` to `$s4`

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j Exit
Else:   sub $s0, $s1, $s2
Exit:
```

Loops (MIPS)

- `while(save[i] == k) i += 1;`
- `i-$s3, k-$s5, save-$s6`

Loops (MIPS)

- `while(save[i] == k) i += 1;`
- `i-$s3, k-$s5, save-$s6`

```
Loop: sll $t1, $s3, 2
```

Loops (MIPS)

- `while(save[i] == k) i += 1;`
- `i-$s3, k-$s5, save-$s6`

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
```

Loops (MIPS)

- `while(save[i] == k) i += 1;`
- `i-$s3, k-$s5, save-$s6`

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw  $t0, 0($t1)
```

Loops (MIPS)

- `while(save[i] == k) i += 1;`
- `i-$s3, k-$s5, save-$s6`

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

Loops (MIPS)

- `while(save[i] == k) i += 1;`
- `i-$s3, k-$s5, save-$s6`

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw  $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s2, $s3, 1
```

Loops (MIPS)

- `while(save[i] == k) i += 1;`
- `i-$s3, k-$s5, save-$s6`

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw  $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s2, $s3, 1
      j    Loop
```

Loops (MIPS)

- while(save[i] == k) i += 1;
- i-$s3, k-$s5, save-$s6

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw  $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s2, $s3, 1
      j     Loop
Exit:
```

Basic blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)
- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Conditional update (MIPS)

- **MIPS instruction – set on less than** `slt rd, rs, rt`
  - `slt $t0, $s3, $s4`
  - `$t0=1` **if** `$s3 < $s4` **holds**
- **Set on less than (immediate) —** `slti $t0, $s2, 10`
- **Can be used in combination with** `bne, beq`

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

Branch instruction design (MIPS)

- There is no `blt, bge` etc in MIPS
- Hardware for $<, \geq, \ldots$ is slower than $=, \neq$
  - Combining with branch involves more work per instruction, require a slower clock
  - All instructions are penalized
- `bne, beq` are the common case
- This is a good design practice

Signed and unsigned comparison (MIPS)

- $s0=1111 1111 1111 1111
- $s1=0000 0000 0000 0001
- **What will be the outcome of the following?**

```
slt $t0, $s0, $s1 ; signed comparison
sltu $t1, $s0, $s1 ; unsigned comparison
```

Encoding of branch instruction (MIPS)

opcode	rs	rt	address
6 bits	5 bits	5 bits	16 bits

Encoding of branch instruction (MIPS)

opcode	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- **PC relative addressing**
  - **Target address = PC + offset $\times$ 4**
  - **PC already incremented by 4 this time**

Encoding of jump instruction (MIPS)

opcode	address
6 bits	26 bits

- **Final address is generated based on PC and address**

Target address example (MIPS)

Label	Instruction	Address						
Loop:	sll $t1, $s3, 2	80000	0	0	19	9	4	0
	add $t1, $t1, $s6	80004	0	9	22	9	0	32
	lw $t0, 0($t1)	80008	35	9	8	0		
	bne $t0, $s5, Exit	80012	5	8	21	2		
	addi $s3, $s3, 1	80016	8	19	19	1		
	j Loop	80020	2	20000				
Exit:		80024						

Branching far away (MIPS)

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example:

```
beq $s0, $s1, L1
```

gets converted to

```
bne $s0, $s1, L2
j L1
L2: ...
```

Procedures

- Put the parameters in a place where procedure can access
- Transfer control to procedure
- Acquire storage resources needed for it
- Perform desired task
- Put return value in a place where the calling program can access it
- Return control to the point of origin

Register usage in MIPS

- `$a0--$a3` : **arguments (registers 4–7)**
- `$v0, $v1` : **result values (registers 2 & 3)**
- `$t0--$t9` : **temporary registers, can be overwritten by callee**
- `$s0--$s7` : **saved registers, must be saved / restored by callee**
- `$gp` : **global pointer for static data (register 28)**
- `$sp` : **stack pointer (register 29)**
- `$fp` : **frame pointer (register 30)**
- `$ra` : **return address (register 31)**

Procedure call instruction (MIPS)

- **Procedure call: *jump and link* —** `jal ProcedureLabel`
  - **Address of following instruction put in** `$ra`
  - **Jumps to target address**
- **Procedure return: *jump register* —** `jr $ra`
  - **Copies** `$ra` **to program counter**
  - **Can also be used for computed jumps (example case/switch statements)**

Example (MIPS)

```
int leaf_examp(int g, int h, int i, int j)
{
      int f;
      f = (g+h)-(i+j);
      return f;
}
```

- **Arguments** g,h,i,j **in** $a0, $a1, $a2, $a3
- f **is in** $s0 **(hence need to save $s0 on stack)**
- **Result is in** $v0

Example (MIPS)

```
Leaf_examp:
```

```
int leaf_examp(int g, int h, int i, int j)
{
     int f;
     f = (g+h)-(i+j);
     return f;
}
```

```
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
```

- **Arguments** `g,h,i,j` **in** `$a0, $a1, $a2, $a3`
- `f` **is in** `$s0` **(hence need to save $s0 on stack)**
- **Result is in** `$v0`

Example (MIPS)

```
int leaf_examp(int g, int h, int i, int j)
{
     int f;
     f = (g+h)-(i+j);
     return f;
}
```

```
Leaf_examp:
addi $sp, $sp, -12
sw   $s0, 0($sp)
sw   $t0, 4($sp)
sw   $t1, 8($sp)
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
```

- **Arguments** `g,h,i,j` **in** `$a0, $a1, $a2, $a3`
- `f` **is in** `$s0` **(hence need to save $s0 on stack)**
- **Result is in** `$v0`

Example (MIPS)

```
int leaf_examp(int g, int h, int i, int j)
{
      int f;
      f = (g+h)-(i+j);
      return f;
}
```

```
Leaf_examp:
addi $sp, $sp, -12
sw   $s0, 0($sp)
sw   $t0, 4($sp)
sw   $t1, 8($sp)
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $zero
```

- **Arguments** `g,h,i,j` **in** `$a0, $a1, $a2, $a3`
- `f` **is in** `$s0` **(hence need to save $s0 on stack)**
- **Result is in** `$v0`

Example (MIPS)

```
int leaf_examp(int g, int h, int i, int j)
{
     int f;
     f = (g+h)-(i+j);
     return f;
}
```

- **Arguments** `g,h,i,j` **in** `$a0, $a1, $a2, $a3`
- `f` **is in** `$s0` **(hence need to save $s0 on stack)**
- **Result is in** `$v0`

```
Leaf_examp:
addi $sp, $sp, -12
sw   $s0, 0($sp)
sw   $t0, 4($sp)
sw   $t1, 8($sp)
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $zero
lw   $s0, 0($sp)
lw   $t0, 4($sp)
lw   $t1, 8($sp)
addi $sp, $sp, 12
```

Example (MIPS)

```
int leaf_examp(int g, int h, int i, int j)
{
    int f;
    f = (g+h)-(i+j);
    return f;
}
```

- **Arguments** `g,h,i,j` **in** `$a0, $a1, $a2, $a3`
- `f` **is in** `$s0` **(hence need to save $s0 on stack)**
- **Result is in** `$v0`

```
Leaf_examp:
addi $sp, $sp, -12
sw   $s0, 0($sp)
sw   $t0, 4($sp)
sw   $t1, 8($sp)
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $zero
lw   $s0, 0($sp)
lw   $t0, 4($sp)
lw   $t1, 8($sp)
addi $sp, $sp, 12
jr   $ra
```

Example: Improved version (MIPS)

```
int leaf_examp(int g, int h, int i, int j)
{
      int f;
      f = (g+h)-(i+j);
      return f;
}
```

```
Leaf_examp:
addi $sp, $sp, -4
sw   $s0, 0($sp)
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $zero
lw   $s0, 0($sp)
addi $sp, $sp, 4
jr   $ra
```

- **Arguments** `g,h,i,j` **in** `$a0, $a1, $a2, $a3`
- `f` **is in** `$s0` **(hence need to save $s0 on stack)**
- **Result is in** `$v0`

Non-leaf procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Nested procedure (MIPS)

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

- **Argument n is in** `$a0`
- **Result is in** `$v0`

Nested procedure (MIPS)

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

```
fact:


     slti $t0, $a0, 1
     beq  $t0, $zero, L1
```

- **Argument n is in** $a0
- **Result is in** $v0

Nested procedure (MIPS)

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

- **Argument n is in** $a0
- **Result is in** $v0

```
fact:

         slti $t0, $a0, 1
         beq  $t0, $zero, L1

L1:      addi $a0, $a0, -1
         jal   fact
```

Nested procedure (MIPS)

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

- **Argument n is in** $a0
- **Result is in** $v0

```
fact:



      slti $t0, $a0, 1
      beq  $t0, $zero, L1



L1:   addi $a0, $a0, -1
      jal  fact



      mul  $v0, $a0, $v0
```

Nested procedure (MIPS)

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

- **Argument n is in** $a0
- **Result is in** $v0

```
fact:
      addi $sp, $sp, -8
      sw   $ra, 4($sp)
      sw   $a0, 0($sp)
      slti $t0, $a0, 1
      beq  $t0, $zero, L1



L1:   addi $a0, $a0, -1
      jal  fact



      mul  $v0, $a0, $v0
```

Nested procedure (MIPS)

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

- **Argument n is in** `$a0`
- **Result is in** `$v0`

```
fact:
      addi $sp, $sp, -8
      sw   $ra, 4($sp)
      sw   $a0, 0($sp)
      slti $t0, $a0, 1
      beq  $t0, $zero, L1

L1:   addi $a0, $a0, -1
      jal  fact
      lw   $a0, 0($sp)
      lw   $ra, 4($sp)
      addi $sp, $sp, 8
      mul  $v0, $a0, $v0
```

Nested procedure (MIPS)

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

- **Argument n is in** `$a0`
- **Result is in** `$v0`

```
fact:
      addi $sp, $sp, -8
      sw   $ra, 4($sp)
      sw   $a0, 0($sp)
      slti $t0, $a0, 1
      beq  $t0, $zero, L1
      addi $v0, $zero, 1
      addi $sp, $sp, 8
      jr   $ra
L1:   addi $a0, $a0, -1
      jal  fact
      lw   $a0, 0($sp)
      lw   $ra, 4($sp)
      addi $sp, $sp, 8
      mul  $v0, $a0, $v0
```

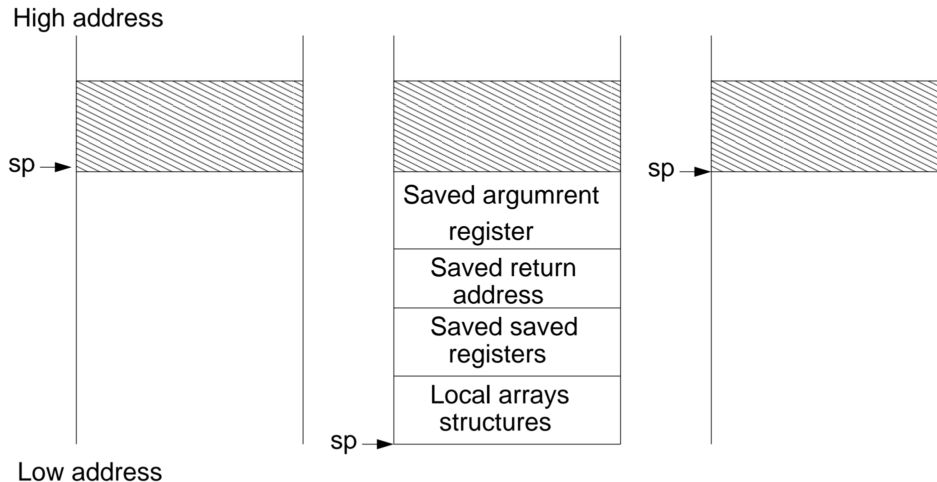
Nested procedure (MIPS)

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

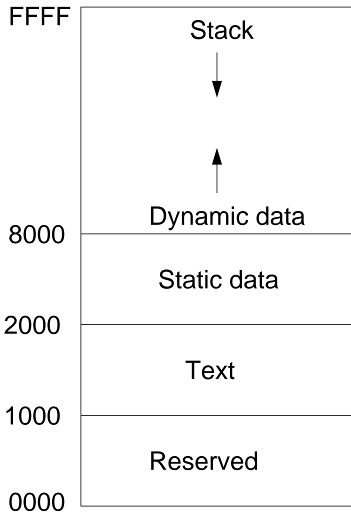
- **Argument n is in** `$a0`
- **Result is in** `$v0`

```
fact:
      addi $sp, $sp, -8
      sw   $ra, 4($sp)
      sw   $a0, 0($sp)
      slti $t0, $a0, 1
      beq  $t0, $zero, L1
      addi $v0, $zero, 1
      addi $sp, $sp, 8
      jr   $ra
L1:   addi $a0, $a0, -1
      jal  fact
      lw   $a0, 0($sp)
      lw   $ra, 4($sp)
      addi $sp, $sp, 8
      mul  $v0, $a0, $v0
      jr   $ra
```

Stack allocation



Memory allocation for data & program



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters (95 graphic, 33 control)
  - Latin-1: 256 characters (ASCII, 96 more graphic character)
- Unicode: 32-bit character set
  - Used in Jaca, C++ wide characters
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable length encoding

Byte/Halfword operations

- Could use bitwise operations
- MIPS has byte/halfword load/store
  - `lb rt, offset(rs)` `lh rt, offset(rs)` : Sign extend to 32 bits in `rt`
  - `lbu rt, offset(rs)` `lhu rt, offset(rs)` : Zero extend to 32 bits in `rt`
  - `sb rt, offset(rs)` `sh rt, offset(rs)` : Store just rightmost byte / halfword

strcpy (MIPS)

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- $a0, y -- $a1, i --$s0
```

strcpy (MIPS)

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- $a0, y -- $a1, i --$s0
```

```
strcpy:
        addi $sp, $sp, -4
        sw   $s0, 0($sp)
        add $s0, $zero, $zero
L1:     add $t1, $s0, $a1
        lbu $t2, 0($t1)
        add $t3, $s0, $a0
        sb $t2, 0($t3)
        beq $t2, $zero, L2
        addi $s0, $s0, 1
        j L1
L2:     lw $s0, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

32-bit constants (MIPS)

- Most constants are small, 16-bit immediate is sufficient
- For the occasional 32-bit constant – `lui rt, constant`
  - Copies 16-bit constant to left 16 bits of `rt`
  - Clears right 16 bits of `rt` to 0
- Example

```
lhi $s0, 61          0000 0000 0111 1101 0000 0000 0000 0000
ori $s0, $s0, 2304   0000 0000 0111 1101 0000 1001 0000 0000
```

Addressing mode (MIPS)

- **Immediate -** `add rt, rs, imm ; rt=rs+imm`
- **Register -** `add rd, rs, rt ; rd=rs+rt`
- **Base addressing -** `lw rt, base, offset ; rt=M[base+offset]`
- **PC relative -** `beq r1, r2, 1000 ; addr = PC + 1000`
- **Pseudodirect addressing -** `j offset; addr = f(PC, addr)`

Addressing mode (ARM)

- **Immediate –** `ADD r2, r0, #5 ; r2=r0+5`
- **Register –** `ADD r2, r0, r1 ; r2=r0+r1`
- **Scaled Register –** `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`
- **PC relative –** `BEQ 1000 ; addr = PC + 1000`
- **Immediate offset –** `LDR r2, [r0,#8]`
  - `r2 = M[r0 + 8]`
- **Register offset –** `LDR r2, [r0,r1]`
  - `r2 = M[r0 + r1]`
- **Scaled register offset –** `LDR r2, [r0,r1,LSL #2]`
  - `r2 = M[r0 + (r1<<2)]`

Addressing mode (ARM)

- **Immediate offset pre-index –** `LDR r2, [r0,#4]!`
  - `r2 = M[ r0 + 4]`
  - `r0 = r0 + 4`
- **Immediate offset post-index –** `LDR r2, [r0], #4`
  - `r2 = M[r0]`
  - `r0 = r0 + 4`
- **Register offset pre-index –** `LDR r2, [r0,r1]!`
  - `r2 = M[r0+r1]`
  - `r0 = r0 + r1`
- **Register offset post-index –** `LDR r2, [r0], r1`
  - `r2 = M[r0]`
  - `r0 = r0 + r1`

Addressing mode (ARM)

- **Scaled register offset pre-index –** `LDR r2, [r0, r1, LSL #2]!`
  - `r2 = M[ r0 + (r1<<2)]`
  - `r0 = r0 + (r1<<2)`

Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1    → add $t0, $zero, $t1

blt $t0, $t1, L  → slt $t2, $t0, $t1
                   bne $t2, $zero, L
```

Arrays vs. Pointer

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory address
  - Can avoid indexing complexity

Arrays vs. Pointer

```
void clear1(int array[], int n)
{
   int i;
   for(i=0; i<n; i++)
     array[i]=0;
}

void clear2(int *array, int n)
{
   int *p;
   for(p=&array[0]; p<array[n]; p++)
     *p=0;
}
```

Arrays vs. Pointer (MIPS)

```
void clear1(int array[], int n)
{
    int i;
    for(i=0; i<n; i++)
      array[i]=0;
}

array -- R0, n -- R1
i -- R2, zero -- R3
```

```
move $t0, $zero
loop1: sll $t1, $t0, 2
sw $zero, 0($t2)
addi $t0, $t0, 1
slt $t3, $t0, $a1
bne $t3, $zero, loop1
```

Arrays vs. Pointer (MIPS)

```
void clear2(int *array, int n){
    int *p;
    for(p=&array[0];
      p<array[n];
      p++)
      *p=0;
}
```

```
array -- R0, n -- R1
p -- R2, zero -- R3
arraySize -- R12
```

```
move $t0, $a0
sll $t1, $a1, 2
add $t2, $a0, $t1
loop2: sw $zero, 0($t0)
addi $t0, $t0, 4
slt $t3, $t0, $t2
bne $t3, $zero, loop2
```

Arrays vs. Pointer (MIPS)

```
clear1:

move $t0, $zero
loop1: sll $t1, $t0, 2
sw $zero, 0($t2)
addi $t0, $t0, 1
slt $t3, $t0, $a1
bne $t3, $zero, loop1
```

```
clear2:

move $t0, $a0
sll $t1, $a1, 2
add $t2, $a0, $t1
loop2: sw $zero, 0($t0)
addi $t0, $t0, 4
slt $t3, $t0, $t2
bne $t3, $zero, loop2
```

Synchronization (MIPS)

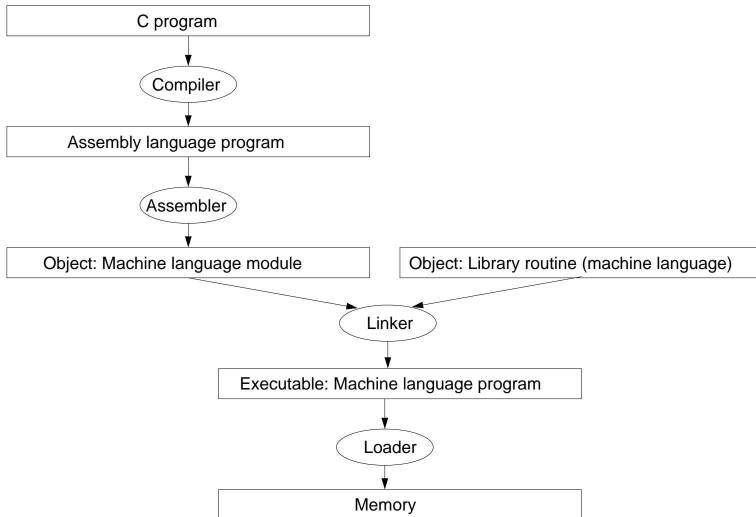
- Two processes access the same memory location
  - P1 writes, then P2 reads
  - Data race if P1 and P2 do not synchronize, results will depend on the order of execution
- Hardware support is required
  - Mutual exclusion
  - Atomic read/write
  - Critical block
  - Atomic exchange or an atomic pair of instructions

Synchronization (MIPS)

- **Load linked:** `ll rt, offset(rs)`
- **Store conditionals:** `sc rt, offset(rs)`
  - **Succeeds if location is not changed since the `ll`, then returns 1**
  - **Fails if location is changed, then returns 0 in `rt`**
- **Example**

```
try:  add $t0, $zero, $s4
      ll $t1, 0($s1)
      sc $t0, 0($s1)
      beq $t0, $zero, try
      add $s4, $zero, $t1
```

Translation hierarchy for C



Assembler

- Object file header – Describe the size and position of the other pieces of the object file
- Text segment – Contains the machine language code
- Static data segment – Contains data allocated for the life time of the program
- Relocation information – Identifies instruction and data word that depend on absolute address when the program is loaded into memory.
- Symbol table – Contains the remaining labels that are not defined
- Debugging information – Extra information so as to associate C source file to machine instruction

Linker

- Place code and data module symbolically in memory
- Determine the address of data and instruction labels
- Patch both the internal and external references

Loader

- Reads the executable file header to determine size of text and data segment
- Creates an address space large enough to store text and data
- Copies the instructions and data from the executable file into memory
- Copies the parameters (if any) to the main program onto stack
- Initialize the machine registers and sets the stack pointer to first free location
- Jumps to start-up routine that copies the parameters into the argument registers and calls the main routine.

- Dynamic linking

Example (ARM)

Object file header				Executable file header		
	Name	Procedure A			Text size	300
	Text size	100			Data size	50
	Data size	20		Text segment	Address	Instruction
Text segment	Address	Instruction			1000	LDR r0, -6000(r3)
	0	LDR r0, 0(r3)			1004	BL 92
	4	BL 0				
Data segment	0	(X)			1100	STR r1, 6020(r3)
Relocation information	Address	Instruction type	Dependency		1104	BL -108
	0	LDR	X			
	4	BL	B	Data segment	Address	
Symbol table	Label	Address			2000	(X)
	X	–			...	...
	B	–			2020	(Y)
Object file header						
	Name	Procedure B				
	Text size	200				
	Data size	30				
Text segment	Address	Instruction				
	0	STR r1, 0(r3)				
	4	BL 0				
Data segment	0	(Y)				
Relocation information	Address	Instruction type	Dependency			
	0	STR	Y			
	4	BL	A			
Symbol table	Label	Address				
	Y	–				
	A	–				

Sort

```
void swap(int v[], int k) {
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}

void sort(int v[], int n){
   int i, j;
   for(i=0; i<n; i++){
     for(j=i-1; j>=0 && v[j]>v[j+1]; j--){
       swap(v, j);
     }
   }
}
```