EC-340 Computer Architecture

Lecture 2

Instruction Set Architecture

Dr Hashim Ali

Fall 2017

Department of Computer Science and Engineering
HITEC University Taxila

Instruction Set Architecture - I

Outline

- Instructions for arithmetic
- Instructions to move data
- Instruction for decision making
- Handiling constant operands

Instructions

- Language of the machine
- Primitive compared to HLLs (High-level Languages)
- Easily interpreted by hardware

Instruction Set Design Goals

- Maximise performance
- Minimise cost
- Reduce design time

Example Instruction Set Architecture

MIPS

- Representative of architectures developed since 1980's
- Used by NEC, Nintendo, Silicon Graphics, Sony
- Real architecture but easy to understand

MIPS Arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

- Example:
  - C code: A = B + C
  - MIPS code: add $s0, $s1, $s2

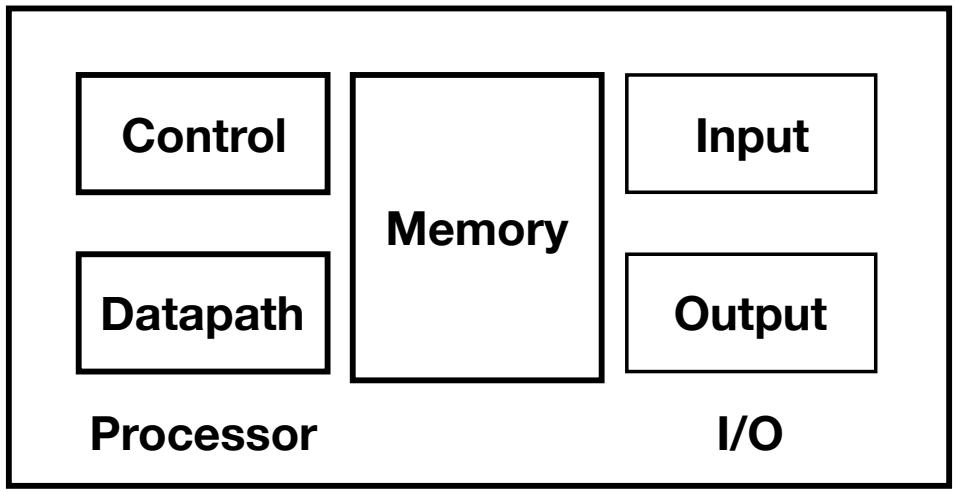
(Associated with variable by compiler)

- Simplicity favours regularity
- Operands must be registers, only 32 registers provided (smaller is faster)
- Expressions need to be broken

C Code	**MIPS Code**
A = B + C + D **E = F - A**	**add $t0, $s1, $s2** **add $s0, $t0, $s3** **add $s4, $s5, $s0**

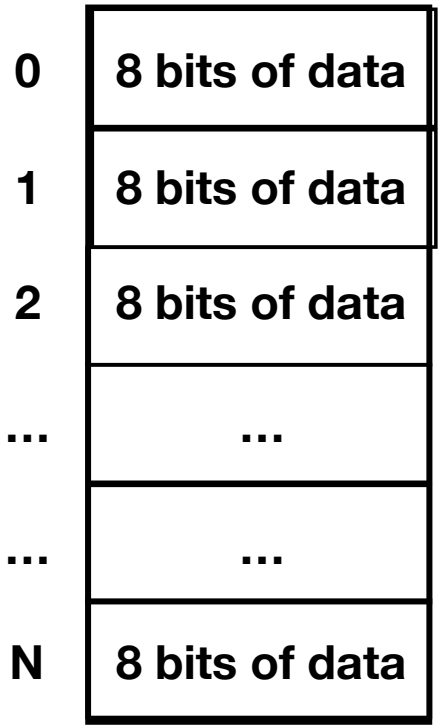
Registers vs. Memory

- Scaler mapped to registers
- Structures, arrays etc. in memory



Memory Organisation

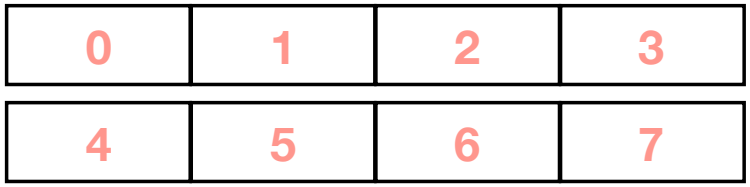
- Viewed as a loarge, single-dimension array, with an address.
- A memory address is an index into the array.
- “Byte addressing” means that the index points a byte of memory.



Words and Bytes

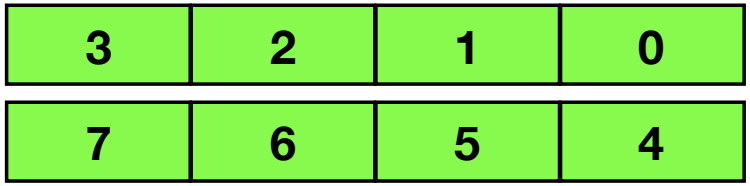
- 2^{32} bytes—byte addresses from 0 to 2^{32}-1
- 2^{30} words—byte addresses 0,4,8,…,2^{32}-4

Big endian byte order



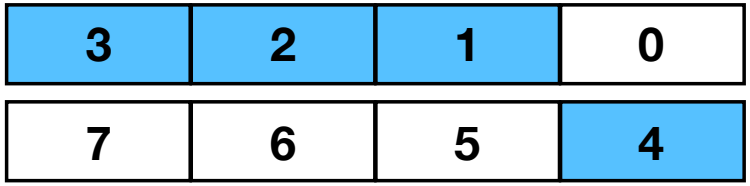
e.g., SPARC Scalable Processor Architecture

Little endian byte order



e.g., Intel Architectures

Non-aligned word



Instructions to Access Memory

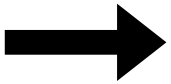
- Load and store instructions
- Example:
  - C code: A[8] = h + A[8];
  - MIPS code:

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

A Simple Example

- What does this code do?

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



```
swap:
  muli $2, $5, 4
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  jr $31
```

Machine Language

- Instructions are 32-bits long
- Registers have numbers 0…31, e.g.,
  - $t0=8, $t1=9, $s0=16, $s1=17 etc.

- Instruction format:

Example: add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
Op	**Rs**	**Rt**	**Rd**	**Shamt**	**Funct**

Machine Language

- lw/sw and the regularity principle?
  - New principle: Good design demands a compromise.
- New format (I-type), other format was R
- Example: lw $t0, 32($s2)

35	18	8	32
Op	Rs	Rt	16-bit number

Control

- Decision making instructions — alter the control flow
- MIPS conditional branch instructions:-
  - bne (branch if not equal) and beq (branch if equal)
- Example:

```
if ( i==j )                bne $s0, $s1, Label
  h = i + j;               add $s3, $s0, $s1
                       Label:
                           …
```

Control

- MIPS unconditional branch instructions:-
  - j (jump) label

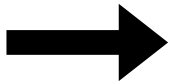
- Example:

```
if ( i!=j )                beq $s4, $s5, Label1
  h = i + j;               add $s3, $s4, $s5
else                       j Label2
  h = i - j;             Label1:
                           sub $s3, $s4, $s5
                         Label2:
                           …
```

Comparison — less or greater

- slt: set-if-less-than

slt $t0, $s1, $s2



```
if $s1 < $s2 then
  $t0 = 1
else
  $t0 = 0
```

- Can use this instruction to build
  - blt $s1, $s2, Label
  - ble $s1, $s2, Label

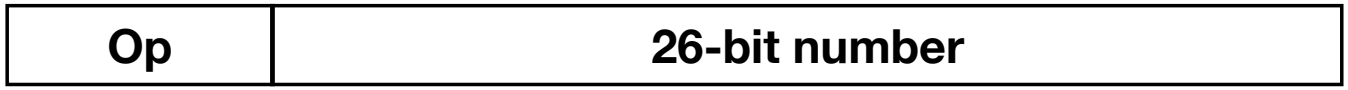
Format of Instructions for Control

- beq, bne

I - Format

- j

J - Format (new)



- slt

R - Format

Constants

- Used quite frequently (50% of operands)
  - e.g., A = A + 5;
  - B = B + 1;
  - C = C - 18;

- Solutions?
  - Data in memory, initialised at load time
  - Put constants within instructions

Constants in MIPS Instructions

addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4

“i” is for “immediate”

I - Format is used

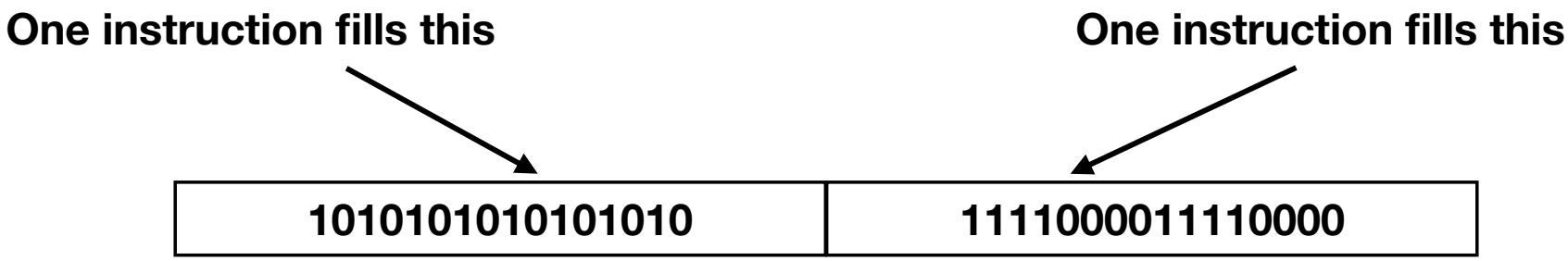
Op	Rs	Rt	16-bit number

A Special Constant

- ‘0’ is a special constant, hard-wired into register $0 (also called $zero)
- add $s2, $s4, $zero
  - Means move from $s4 to $s2

How About Larger Constants?

- To load 32 bit constant into a register, we use two instructions



Loading Larger Constants

- New “load upper immediate” instruction
  - lui $t0, 1010101010101010
- Then get the lower oder bits right, I.e.,
  - ori $t0, $t0, 1111000011110000

1010101010101010	**0000000000000000**
0000000000000000	**1111000011110000**

1010101010101010	**1111000011110000**

Summary of Instructions Learnt

• **Instructions**	• **Format**
add, sub, addi, subi	R, I
and, or, andi, ori	R, I
slt, slti	R, I
beq, bne	I
j	J
lw, sw	I
lui	I

Instruction Set Architecture - II

Outline

- Implementation of loops
- Pointer vs index
- Switch statement
- Addresses in MIPS instructions

Writing a Simple Loop for $\sum_i A[i]$

```
s = 0;
i = 0;
L: s = s + A[i];
    i++;
    if ( i < n )
        goto L;
```

Writing a Simple Loop for $\sum_i A[i]$

```
s = 0;
i = 0;
L: s = s + A[i];

    i++;
    if ( i < n ) goto L;
```

```
move $s0, $zero

move $t0, $zero

L: muli $t1, $t0, 4

    add $t1, $t1, $s1        # $s1=&A[0]

    lw $t2, 0($t1)

    add $s0, $s0, $t2

    addi $t0, $t0, 1

    blt $t0, $s2, L          #$s2=n
```

Pseudo Instruction *blt*

blt r1, r2, label

is equivalent to

Real Instructions

slt $at, r1, r2
bne $at, $zero, label

$at — assembler temporary

Overall Program

```
    .data
A: .space 400
    .text
    la $s1, A              # lui $s1, Aupper; ori $s1, $s1, Alower
    li $s2, 100            # ori $s2, $zero, 100
    Code for input
    Code for computation
    Code for output
```

Improving Code: Pointer vs. Index

```
s = 0;
i = 0;

L: s = s + A[i];

   i++;
   if ( i < n ) goto L;
```

```
s = 0;
i = 0;
p = &A[0];

L: s = s + *p;
   p++;

   i++;
   if ( i < n ) goto L;
```

Improving Code: Pointer vs. Index

```
s = 0;
i = 0;
p = &A[0];

L: s = s + *p;
   p++;

   i++;
   if ( i < n ) goto L;
```

```
move $s0, $zero
move $t0, $zero
la $t1, A

L: lw $t2, 0($t1)
   add $s0, $s0, $t2
   addi $t1, $t1, 4

   addi $t0, $t0, 1
   blt $t0, $s2, L      # $s2 = n
```

Improving Code Further

```
s = 0;
i = 0;
p = &A[0];


L: s = s + *p;
   p++;

   i++;
   if ( i < n ) goto L;
```

```
s = 0;

p = &A[0];
q = p + 100;

L: s = s + *p;
    p++;


   if ( p < q ) goto L;
```

Improving Code Further

```
s = 0;
p = &A[0];
q = p + 100;

L: s = s + *p;


  p++;
  if ( p < q ) goto L;
```

```
move $s0, $zero
la $t1, A
addi $s2, $t1, 400

L: lw $t2, 0($t1)
    add $s0, $s0, $t2

    addi $t1, $t1, 4
    blt $t1, $s2, L
```

Nested Loops: Sorting

```
    p = &A[0];
    q = p + 99;
X: r = p + 1;
Y: main step;
    r++;
    if ( r ≤ q ) goto Y;
    p++;
    if ( p < q ) goto X;
```

```
    la $t1, A
    addi $s2, $t1, 396
X:  addi $t2, $t1, 4
Y:  main step;
        add $t2, $t2, 4
        ble $t2, $s2, Y
        addi $t1, $t1, 4
        blt $t1, $s2, X
```

Sorting — Main Step

Y: if (*p < *r) goto Z;

+p↔ *r;

Z: r++;

```
Y: lw $t3, 0($t1)
   lw $t4, 0($t2)
   blt $t3, $t4, Z

   sw $t3, 0($t2)
   sw $t4, 0($t1)

Z: addi $t2, $t2, 4
```

Implementing SWITCH Statement

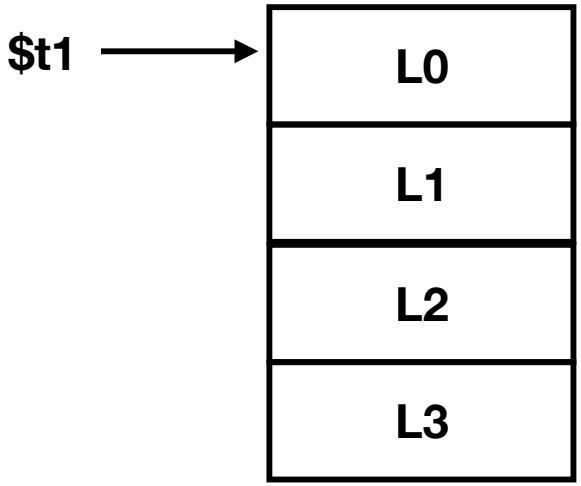
```
switch ( k ) {
    case 0: f = i + j; break;
    case 1: f = g + h; break;
    case 2: f = g - h; break;
    case 3: f = i - j; break;
}
```

Multi-way Branch using Table

```
blt $s5, $zero, Exit        # $s5 has k
bge $s5, $t2, Exit          # $t2 has 4
add $t1, $s5, $s5
add $t1, $t1, $t1           # $t1 gets 4*k
lw $t0, 0($t1)
jr $t0
```

Cases of Switch Statement

```
L0: add $s0, $s3, $s4
    j Exit
L1: add $s0, $s1, $s2
    j Exit
L2: add $s0, $s1, $s2
    j Exit
L3: add $s0, $s3, $s4
Exit
```



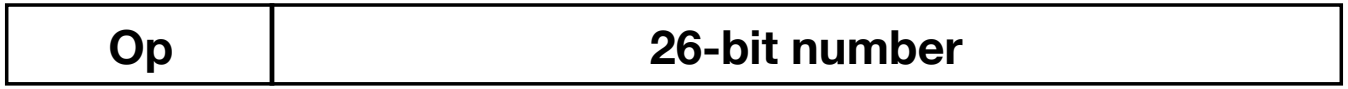
Addresses in MIPS Instructions

- beq, bne, lw, sw

I - Format

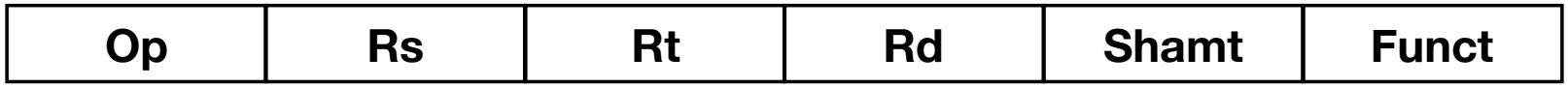
- j

J - Format



- jr

R - Format



Addresses in *lw*, *sw* Instructions

- lw, sw

I - Format

Op	Rs	Rt	16-bit number

- lw r1, C(r2) # r1 = MEM[C+r2]
- sw r1, C(r2) # MEM[C+r2] = r1

- C is a byte offset in range -2^{15} to $2^{15}-1$

Addresses in *beq*, *bne* Instructions

- beq, bne

I - Format

Op	Rs	Rt	16-bit number

- beq r1, r2, C # if (r1 == r2) PC = PC + C + 4
- # else PC = PC + 4
- bne r1, r2, C # if (r1 != r2) PC = PC + C + 4
- # else PC = PC + 4

- C is a word offset in range -2^{15} to $2^{15}-1$

Addresses in *j* Instructions

- beq, bne

J - Format

Op	26-bit number

- PC = PC + 4
- PC[2…27] = instruction[0…25]
- PC[28…31] = PC[28…31]
- PC[0…1] = 00

- Jump within current seguent of 256MB

Addresses in *jr* Instructions

- jr

R - Format

Op	Rs	Rt	Rd	Shamt	Funct

- jr r1 # PC = r1

- Jump anywhere

Summary

- Loops and arrays
- Improving code using pointers
- Implementing switch statement using a table of addresses
- Addressees field in various MIPS instructions

Instruction Set Architecture - III

Summary of Instructions Learnt

• **Instructions**	• **Format**
add, sub, and, or, slt	R
addi, subi, andi, ori, slti	I
beq, bne	I
j	J
lw, sw	I
lui	I
jr	R

Pseudo Instructions

move r1, r2

la r1, address

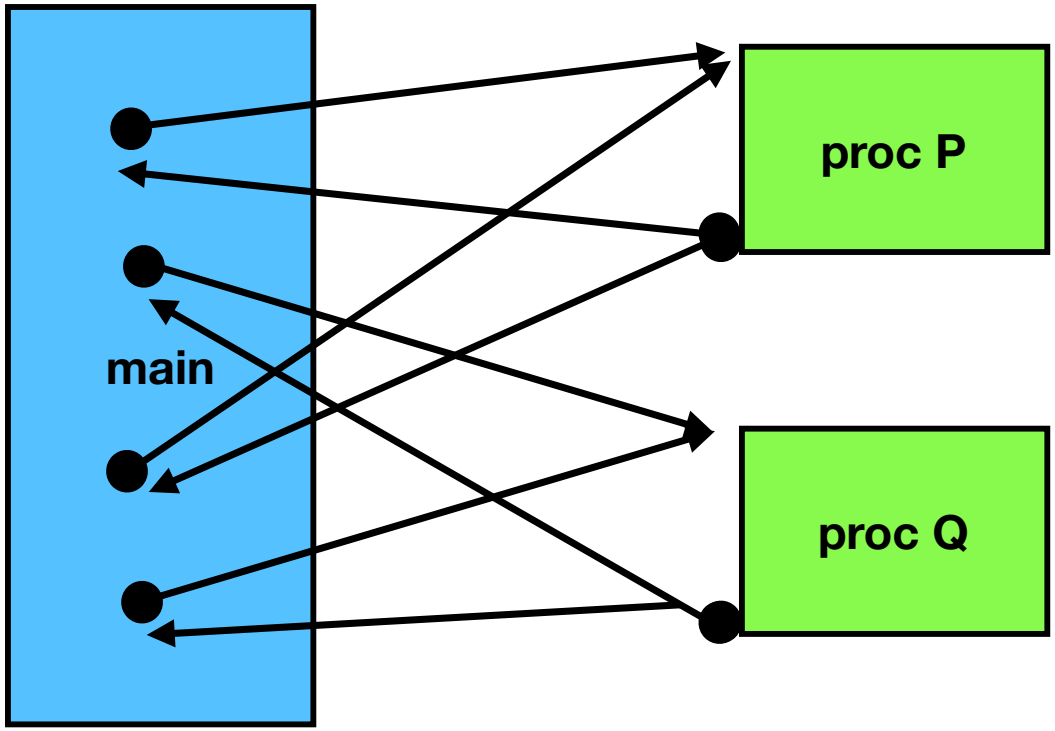
li r1, constant

bgt, bge, blt, ble

Outline

- Procedural abstraction
- Requirements
- Sorting algorithms
- Register use conventions

Procedural Abstractions



What is Required?

- Control flow (call and return)
- Data flow (parameter passing)
- Local and global storage allocation
- Take care of nesting
- Take care of recursion

Control Flow — call

```
    p = &A[0];
    q = p + 99;
X: r = p + 1;
Y: xchg();
   r++;
   if ( r ≤ q ) goto Y;
   p++;
   if ( p < q ) goto X;
```

```
    la $t1, A
    addi $s2, $t1, 396
X: addi $t2, $t1, 4
L: jal xch              # $ra gets PC+4
   add $t2, $t2, 4
   ble $t2, $s2, Y
   addi $t1, $t1, 4
   blt $t1, $s2, X
```

Control Flow — return

```
void xchg() {
Y: if ( *p < *r ) goto Z;

    +p↔ *r;

Z: return;
}
```

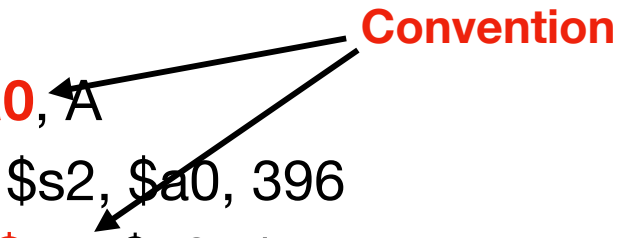
```
xchg:
    lw $t3, 0($t1)
    lw $t4, 0($t2)
    blt $t3, $t4, Z

    sw $t3, 0($t2)
    sw $t4, 0($t1)

Z: jr $ra            # PC gets $ra
```

Parameter Passing through Registers

```
    p = &A[0];
    q = p + 99;
X: r = p + 1;
Y: xchg(p, r);
   r++;
   if ( r ≤ q ) goto Y;
   p++;
   if ( p < q ) goto X;
```



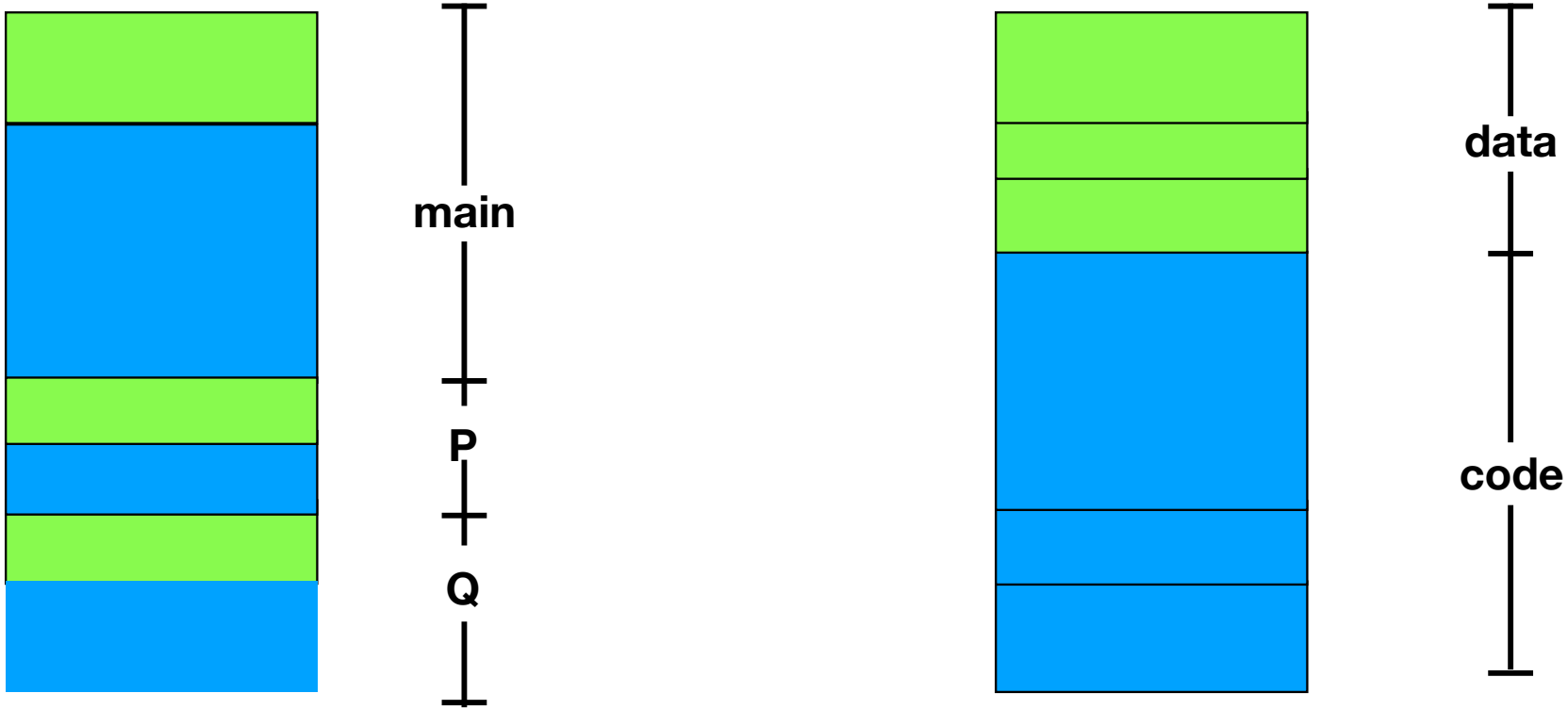

```
    la $a0, A
    addi $s2, $a0, 396
X: addi $a1, $a0, 4
L: jal xch                  # $ra gets PC+4
   add $a1, $a1, 4
   ble $a1, $s2, Y
   addi $a0, $a0, 4
   blt $a0, $s2, X
```

Passing many Parameters

- Convention:
  - Input to procedure—$a0, $a1, $a2, $a3
  - Output from procedure—$v0, $v1

- Additional parameters:
  - Use memory

Local Storage Allocation



Nested Calls

```
P();

jal P
```

```
void P() {
    …Q();…
    return
}

P:  ……
    jal Q…
    jr $ra
```

```
void Q() {
    ……
    return
}

Q: ……
   ……
    jr $ra
```

Nested Call — Example

```
    p = &A[0];
    q = p + 99;
X: r = p + 1;
   min(p,r);
   p++;
  if ( p < q ) goto X;
```

```
void min (int *p, *r) {
     Y: xchg(p, r);
     r++;
     if (r ≤ q) goto Y;
}

void xchg (int *p, *r) {
    if (*p < *r) goto Z;
         +p↔ *r;
    Z: return;
}
```

Nested Call — main

```
    p = &A[0];
    q = p + 99;
X: r = p + 1;
   min(p,r);
   p++;
  if ( p < q ) goto X;
```

```
    la $a0, A
    addi $s2, $a0, 396
X: addi $a1, $a0, 4
   jal min
   add $a0, $a0, 4
   blt $a0, $s2, X
```

Nested Call — proc main

```
void main (int *p, *r) {

    Y: xchg(p, r);
    r++;
    if (r ≤ q) goto Y;
}
```

```
min:
    sw $ra, ra_save
Y:  jal xchg
    addi $a1, $a1, 4
    ble $a1, $s2, Y
    lw $ra, ra_save
    jr $ra
```

Recursive Call — Control Flow

```
void main (int *p, *r) {

    xchg(p, r);
    r++;
    if (r ≤ q)
      min(p,r);
}
```

```
min:
      sw $ra, ra_save
      jal xchg
      addi $a1, $a1, 4
      bgt $a1, $s2, R
      jal min
R:  lw $ra, ra_save
      jr $ra
```

Recursive Call — Control Flow

```
void main (int *p, *r) {

    xchg(p, r);
    r++;
    if (r ≤ q)
      min(p,r);
}
```

```
min:    sw $ra, ra_save      push $ra
        jal xchg
        addi $a1, $a1, 4
        bgt $a1, $s2, R
        jal min              pop $ra
R:      lw $ra, ra_save
        jr $ra
```

Stack and *push/pop* Operations

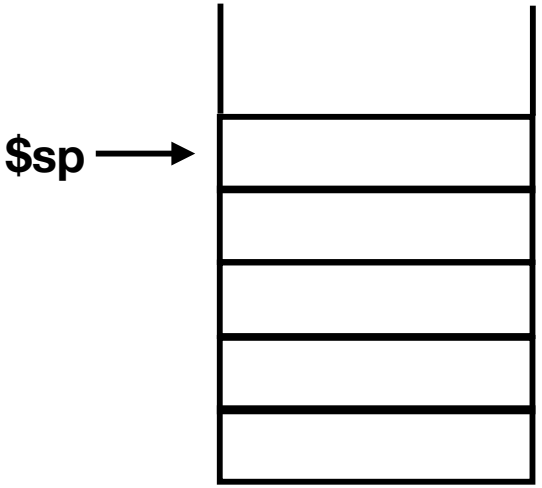
- $sp is the stack pointer

- push $ra

```
addi $sp, $sp, -4
sw $ra, 0($sp)
```

- pop $ra

```
lw $ra, 0($sp)
addi $sp, $sp, 4
```



Recursive Calls — Data Flow

- Stack is used for passing parameters when registers are not sufficient.
- Stack is used for allocating local variables.

Sharing Registers

- Registers which are shared between the caller and callee should be saved at the time of call and restored at the time of return.

- Who does this?

Conventions for Saving Registers

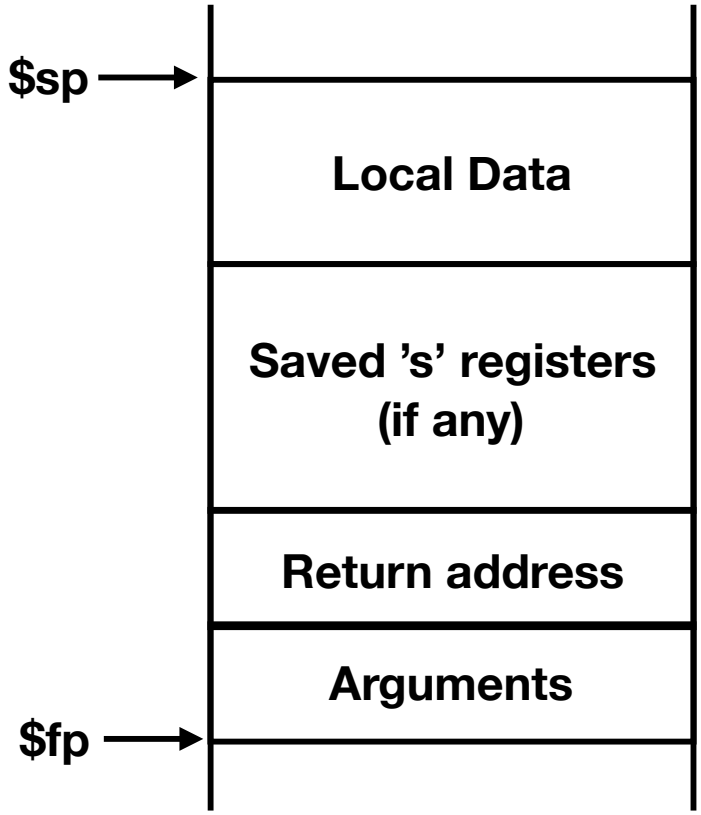
- $s0 to $s7 are called “saved memory”
  - These are preserved across calls
  - Saved by callee, if necessary

- $t0 to $t9 are called “temporary”
  - These are not preserved across calls
  - Saved by callee, if necessary

Register Names and Purpose

Name	Register Number	Usage
$zero	0	The constant value 0
$v0—$v1	2—3	Values of result
$a0—$a3	4—7	Arguments
$t0—$t7	8—15	Temporaries
$s0—$s7	16—23	Saved
$t8—$t9	24—25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

Activation Record/Frame



Summary

- Procedure call and return
- Parameter passing
- Nesting and recursion
- Register use conventions