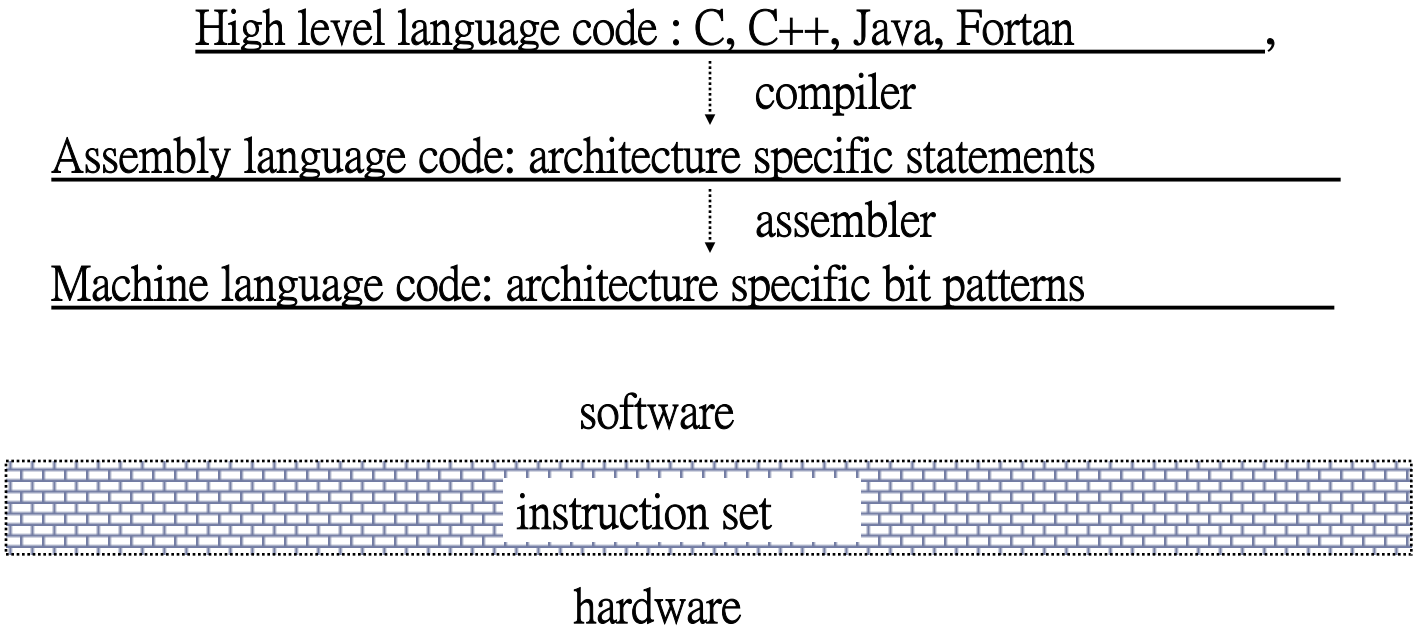
Computer Architecture

Instruction Set Architecture

Instruction Set Architecture

- The instruction set architecture serves as the interface between software and hardware.
- It provides the mechanism by which the software tells the hardware what should be done.



ISA Classes

- The classification is based on internal storage in a processor
  - Stack
  - Accumulator
  - General Purpose Register
    - Register Memory
    - Register Register / Load Store

Why CPU Storage?

- A small amount of storage in the CPU
  - To reduce memory traffic by keeping repeatedly used operands in the CPU
    - Avoid re-referencing memory
    - Avoid having to specify full memory address of the operand
- Simplest Case
  - A machine with 1 cell of CPU storage: the accumulator

ISA Classes: Accumulator

Accumulator :

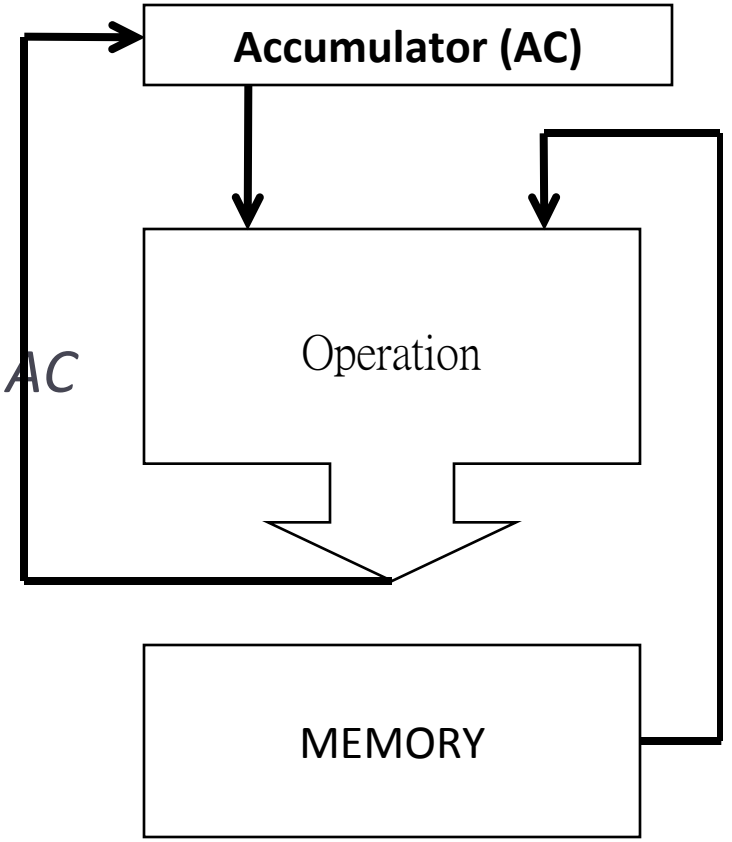
Implicit input & output.

C= A+B?

LOAD A - *Put A in Accumulator*

ADD B - *Add B with AC put result in AC*

STORE C- *Put AC in C*



ISA Classes: Stack

Operate on TOS, put result TOS

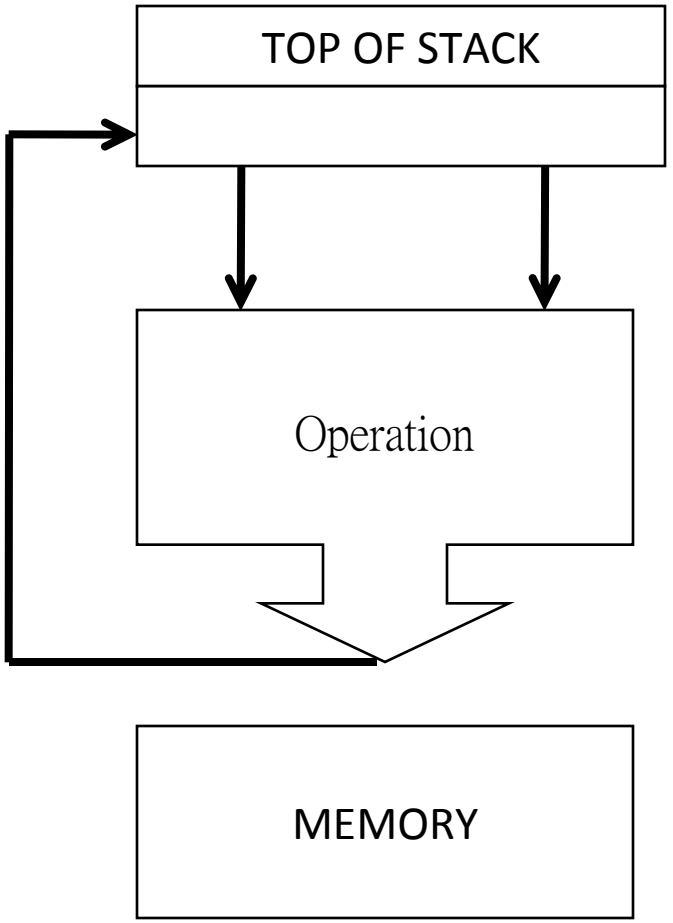
C= A+B?

PUSH A

PUSH B

ADD

POP C



ISA Classes: General Purpose Register

- With stack machines, only the top two elements of the stack are directly available to instructions.
- In general purpose register machines, the CPU storage is organized as a set of registers which are equally available to the instructions

- **1975-present all machines use general purpose registers**

ISA Classes: Register-Memory

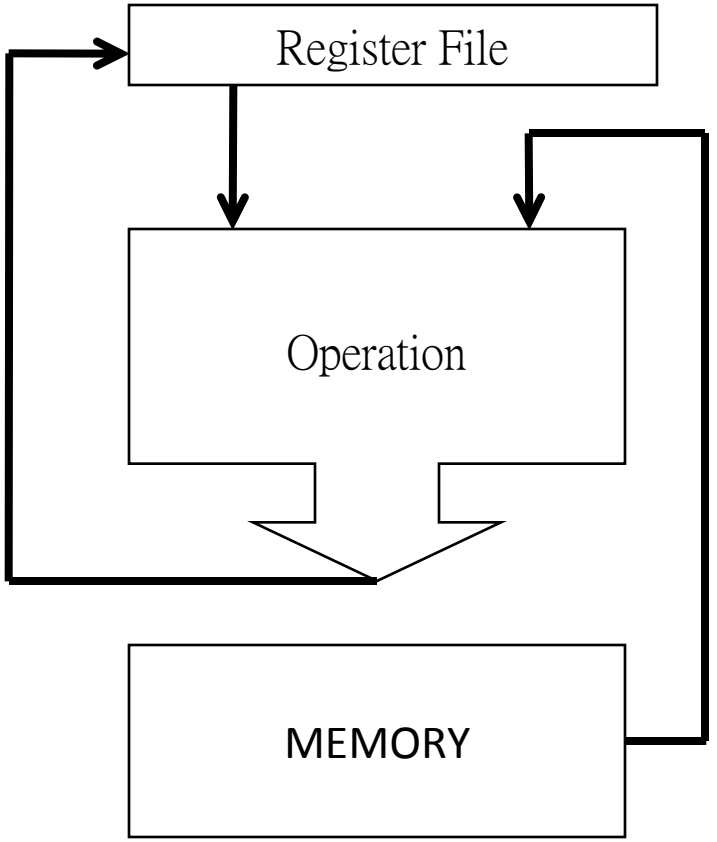
Input, Output: Register or Memory

C= A+B?

LOAD R1, A

ADD R3, R1, B

STORE R3, C



ISA Classes: Register-Register

LOAD/STORE ARCH

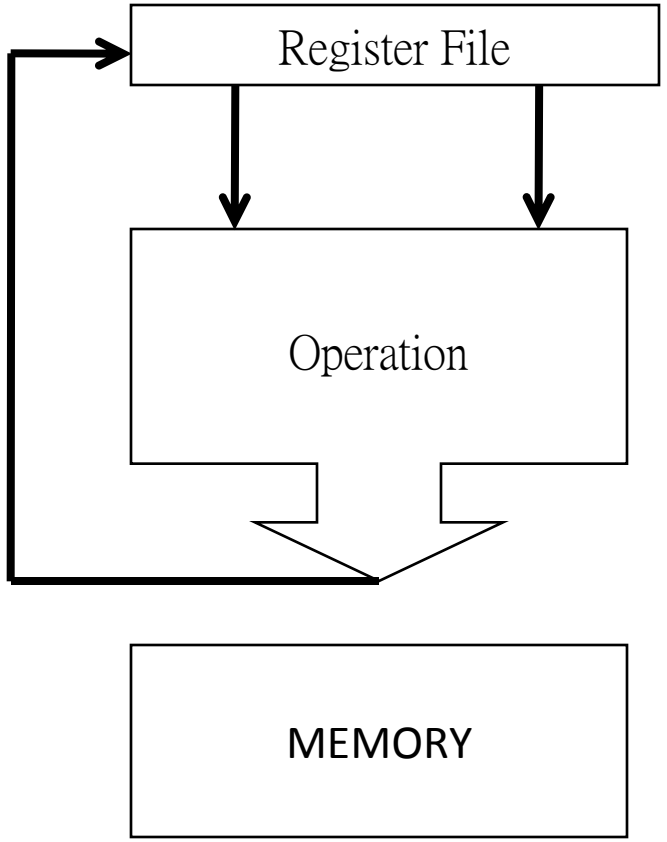
C= A+B?

LOAD R1, A

LOAD R2, B

ADD R3, R1, R2

STORE R3, C



RISC

- Known as Reduced Instruction Set Computer (RISC)
  - Keep the instruction set small and simple, makes it easier to build fast hardware.
  - Let software do complicated operations by composing simpler ones.
- Very popular, used in many products
  - Silicon Graphics, ATI, Cisco, Sony, etc.
- Example: MIPS, PowerPC, SPARC

CISC

- Known as Complex Instruction Set Computer (CISC)
  - Add more and more instructions to new CPUs to do elaborate operations!
- Example: Intel IA-32 , VAX
- Intel IA-32 (*Intel Architecture, 32-bit*)
- First implemented in the Intel 80386 as a 32-bit extension of x86 architecture
- The success of x86 architecture does not contradict the advantage of a RISC instruction set
- Recent 80x86 microprocessors, such as the Pentium 4, use hardware to translate from 80x86 instructions to RISC-like instructions and then execute the translated operations inside the chip

MIPS

- MIPS: “Microprocessor Without Interlocked Pipeline Stages”
- A simple load-store instruction set
- Design for pipelining efficiency, including a fixed instruction set encoding

- MIPS provides a good architectural model for study
- Popularity of this type processor
- Easy architecture to Understand

MIPS General-Purpose Registers

- 32 General Purpose Registers (GPRs)
  - Assembler uses the dollar notation to name registers
    - $0 is register 0, $1 is register 1, …, and $31 is register 31
  - All registers are 32-bit wide in MIPS32
  - Register $0 is always zero
    - Any value written to $0 is discarded
- Software conventions
  - Software defines names to all registers
    - To standardize their use in programs
  - Example: $8 - $15 are called $t0 - $t7
    - Used for temporary values

$0 = $zero	$16 = $s0
$1 = $at	$17 = $s1
$2 = $v0	$18 = $s2
$3 = $v1	$19 = $s3
$4 = $a0	$20 = $s4
$5 = $a1	$21 = $s5
$6 = $a2	$22 = $s6
$7 = $a3	$23 = $s7
$8 = $t0	$24 = $t8
$9 = $t1	$25 = $t9
$10 = $t2	$26 = $k0
$11 = $t3	$27 = $k1
$12 = $t4	$28 = $gp
$13 = $t5	$29 = $sp
$14 = $t6	$30 = $fp
$15 = $t7	$31 = $ra

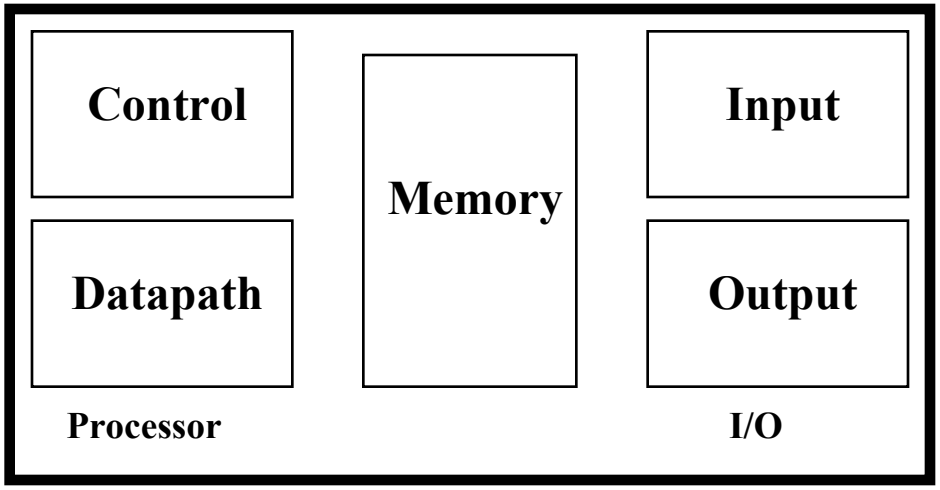
MIPS Register Conventions

- Assembler can refer to registers by name or by number
  - It is easier for you to remember registers by name
  - Assembler converts register name to its corresponding number

Name	Register	Usage
$zero	$0	Always 0 (forced by hardware)
$at	$1	Reserved for assembler use
$v0 – $v1	$2 – $3	Result values of a function
$a0 – $a3	$4 – $7	Arguments of a function
$t0 – $t7	$8 – $15	Temporary Values
$s0 – $s7	$16 – $23	Saved registers (preserved across call)
$t8 – $t9	$24 – $25	More temporaries
$k0 – $k1	$26 – $27	Reserved for OS kernel
$gp	$28	Global pointer (points to global data)
$sp	$29	Stack pointer (points to top of stack)
$fp	$30	Frame pointer (points to stack frame)
$ra	$31	Return address (used by jal for function call)

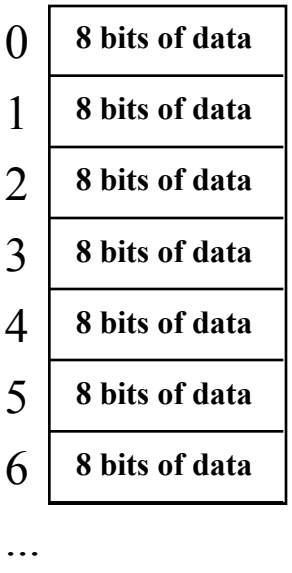
Registers vs. Memory

- Arithmetic instructions operands must be registers,
  - only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?



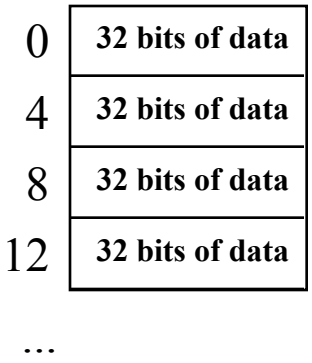
Memory Organization

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



Memory Organization

- Bytes are nice, but most data items use larger "words"
- MIPS provides **lw/lh/lb** and **sw/sh/sb** instructions
- For MIPS, a word is 32 bits or 4 bytes.



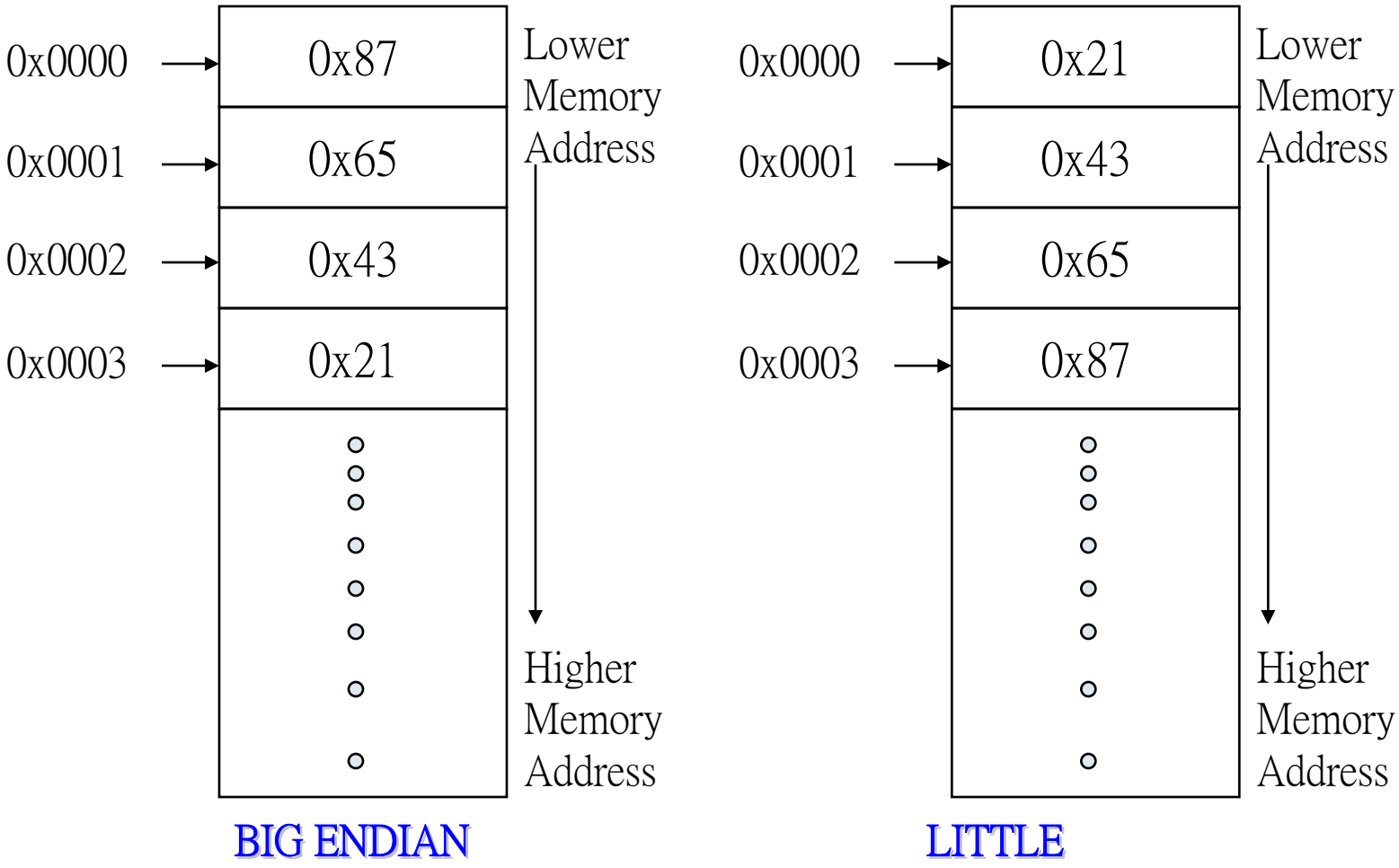
Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$

Endianness [defined by Danny Cohen 1981]

- Byte ordering — How is a multiple byte data word stored in memory
  - Big Endian
    - Most significant byte of a multi-byte word is stored at the lowest memory address
    - Address of most significant byte = word address
    - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
  - Little Endian
    - Least significant byte of a multi-byte word is stored at the lowest memory address
    - Address of least significant byte = word address
    - e.g. Intel x86, DEC Vax, DEC Alpha (Windows NT)

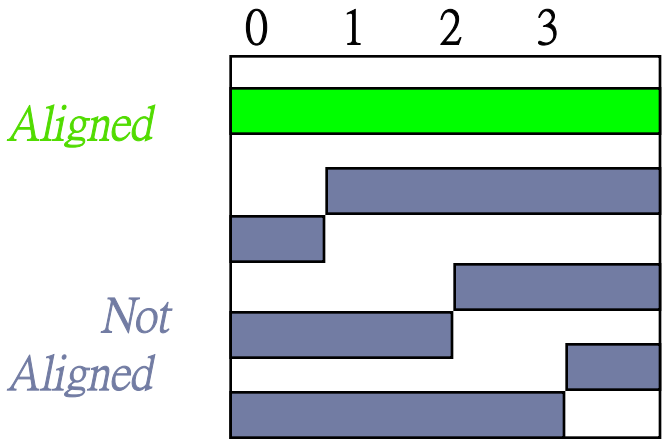
Example of Endian



Store 0x87654321 at address 0x0000, byte-addressable

Alignment

- Alignment: require that objects fall on address that is multiple of their size.



MIPS Arithmetic Instructions

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

Example:

C code: A = B + C

MIPS code: add $s0, $s1, $s2

Design Principle 1: Simplicity favors regularity

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

MIPS Arithmetic Instructions

- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

C code:A = B + C + D;
E = F - A;

MIPS code:
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0

- Operands must be registers, only 32 registers provided
- All memory accesses are accomplished via loads and stores
  - A common feature of RISC processors
- What if we do not have enough registers? *register spilling*

MIPS Memory Instructions

- Load and store instructions
- Example:

C code: long A[100];

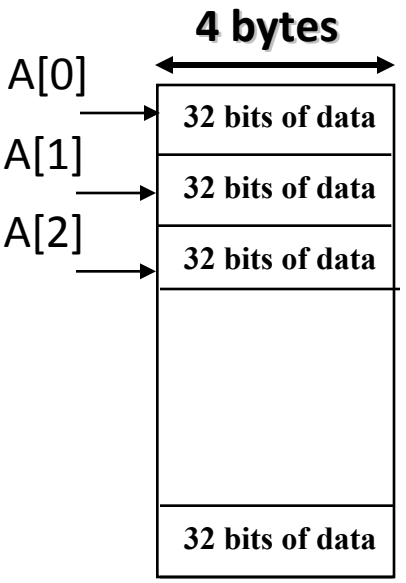
A[9] = h + A[8];

MIPS code: lw $t0, 32($s3) #load word

add $t0, $s2, $t0

sw $t0, 36($s3)

index base register



- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Practice: SWAP

- Swapping words

```
temp = v[0]
v[0] = v[1];
v[1] = temp;
```

- $s2 has the base address of the array v

```
swap:
    lw $t0, 0($s2)
    lw $t1, 4($s2)
    sw $t0, 4($s2)
    sw $t1, 0($s2)
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!
- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

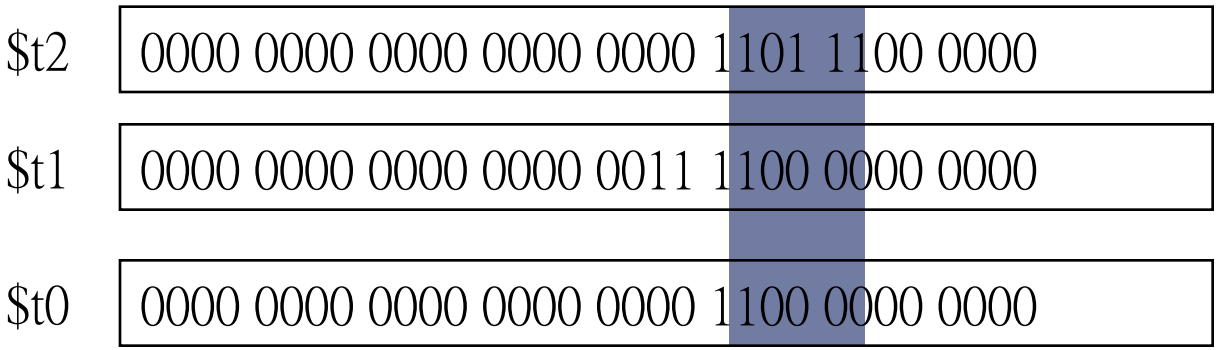
Shift Operations

- Shift left logical
  - Shift left and fill with 0 bits
  - sll by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - srl by *i* bits divides by 2^i (unsigned only)
- Example: **sll $t0, $s2, 3**
  - $S2: 0110 0000 0000 0000 1100 1000 0000 1111
  - $t0: 0000 0000 0000 0110 0100 0000 0111 1000

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

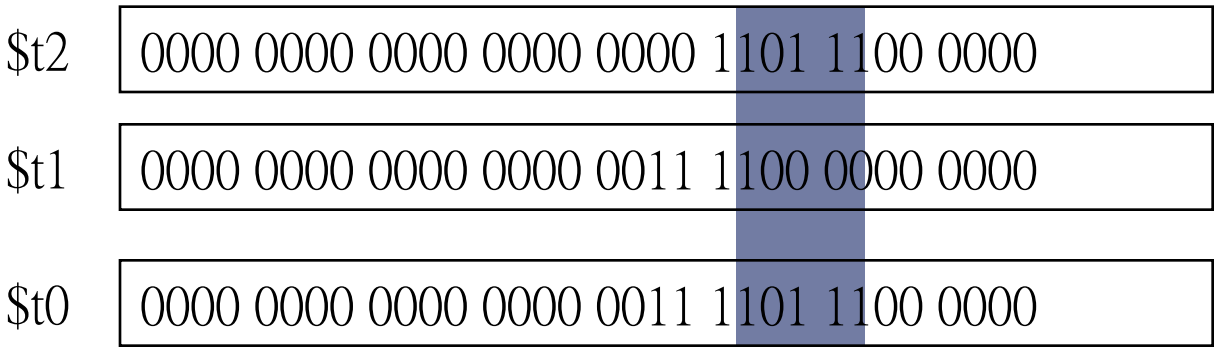
and $t0, $t1, $t2



OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

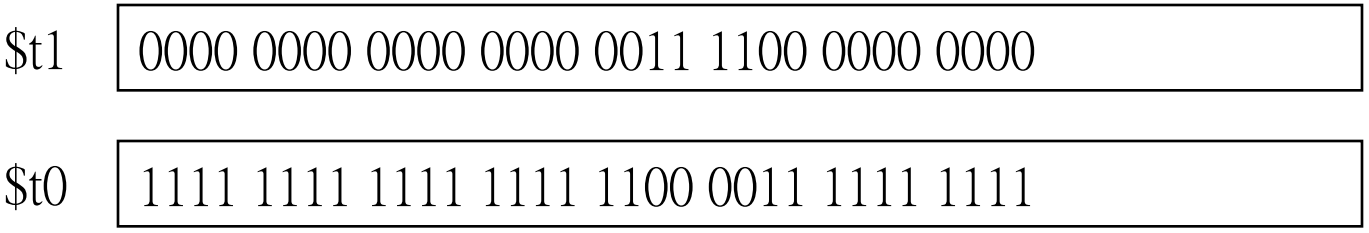
or $t0, $t1, $t2



NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

nor $t0, $t1, $zero



Constants Or Immediate Operands

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;

  C = C - 18;

- Solutions?
  - Put 'typical constants' in memory and load them.
  - Create hard-wired registers (like $zero) for constants like one
  - Use immediate values
- MIPS Instructions:

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

Constants Or Immediate Operands

- No subtract immediate instruction
  - Just use a negative constant

    addi $s2, $s1, -1

- Hardwired values useful for common operations
  - E.g., move between registers

    add $t2, $s1, $zero

- Design Principle 3: Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Machine Language vs. Assembly Language

- Assembly Code
  - a collection of instructions expressed in “textual” format e. g. Add r1, r2, r3
  - much easier than writing down numbers
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code

- Machine Code
  - a collection of instructions encoded in binary format
  - directly consumable by the hardware

Machine Language

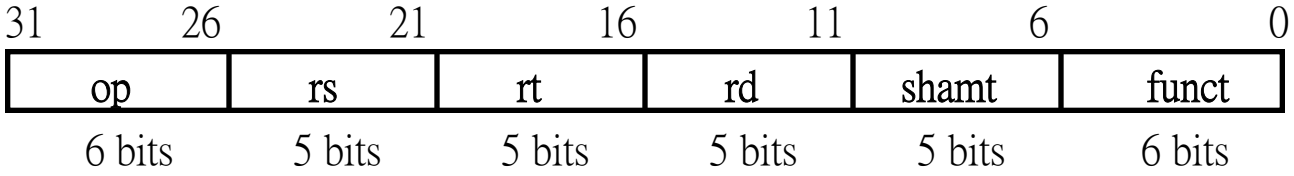
- Instructions, like registers and words of data, are 32 bits long
  - Example: add $t0, $s1, $s2
  - registers have numbers, $t0=8, $s1=17, $s2=18

- Instruction Format:

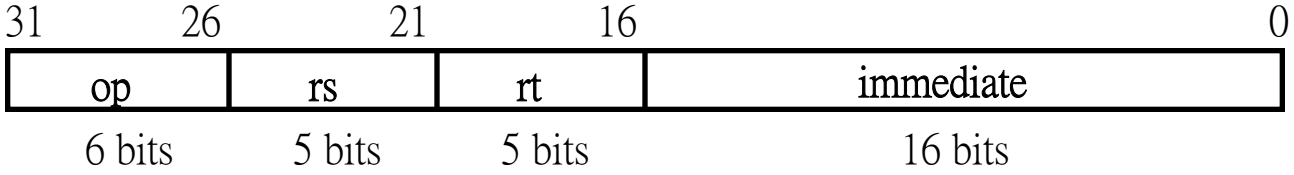
000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

MIPS Instruction Format

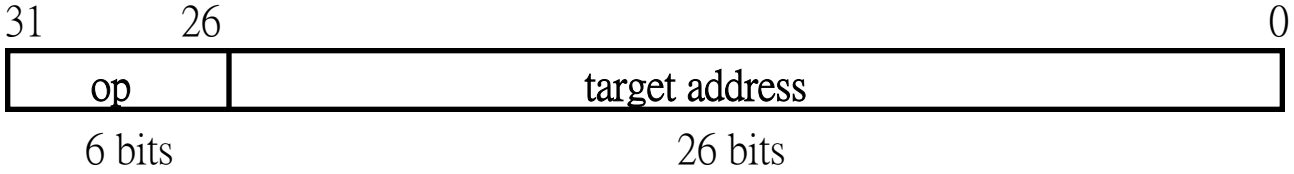
- **All MIPS instructions are 32 bits long. 3 formats:**
  - R-type



  - I-type

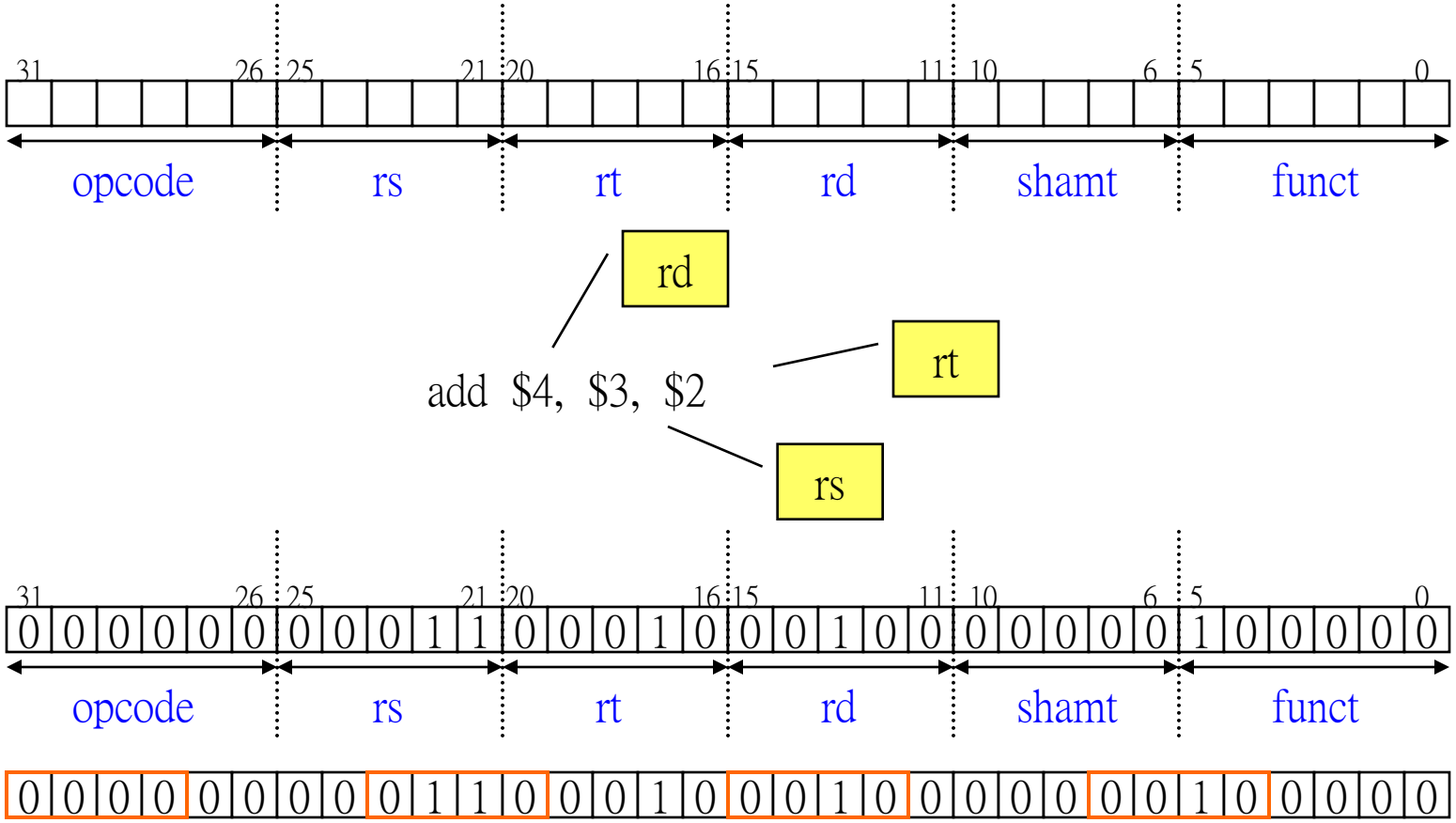


  - J-type



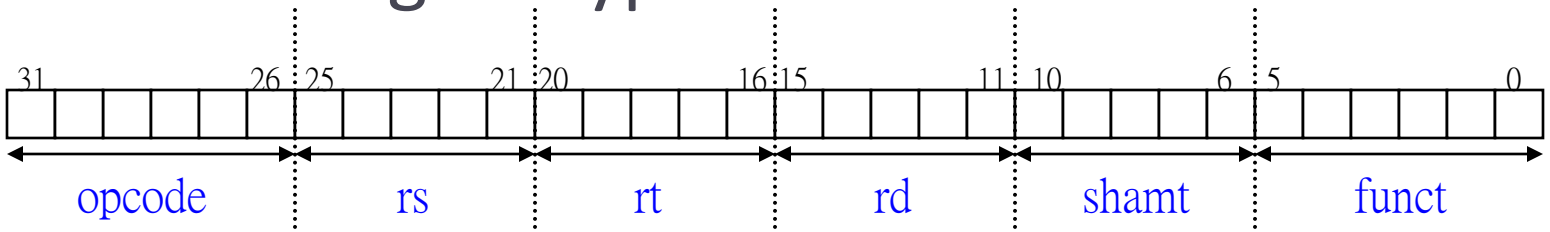
- **The different fields are:**
  - **op:** operation (“opcode”) of the instruction
  - **rs, rt, rd:** the source and destination register specifiers
  - **shamt:** shift amount
  - **funct:** selects the variant of the operation in the “op” field
  - **address / immediate:** address offset or immediate value
  - **target address:** target address of jump instruction

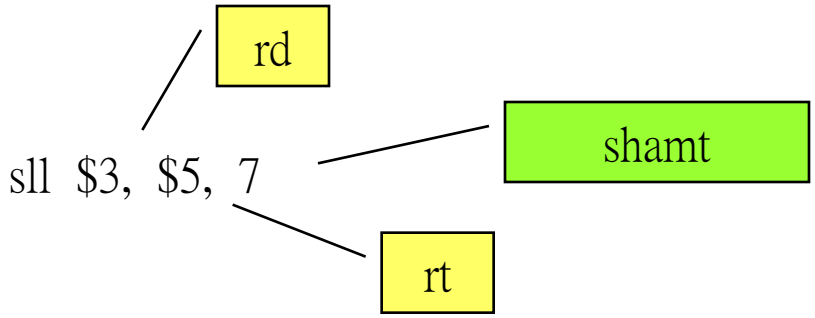
MIPS Encoding: R-Type

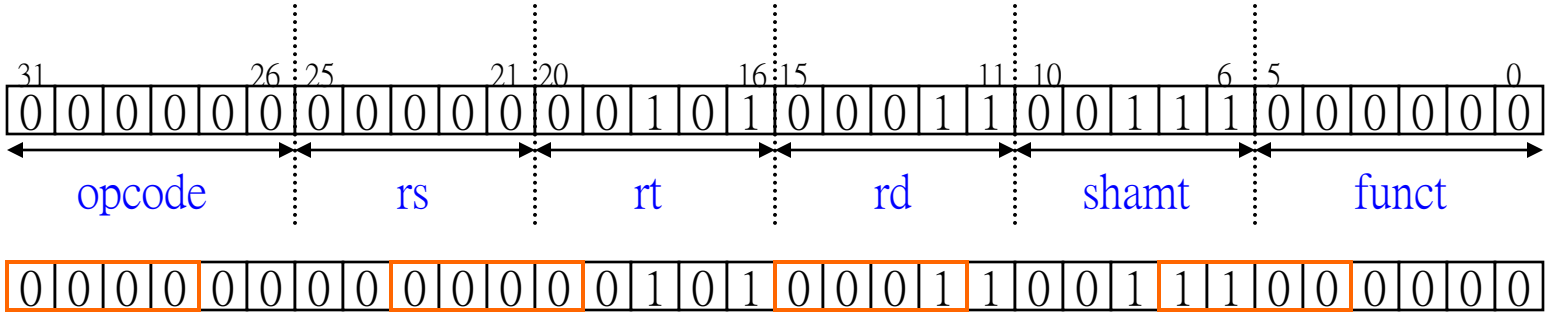


Encoding = 0x00622020

MIPS Encoding: R-Type







Encoding = 0x000519C0

Machine Language

- Consider the load-word and store-word instructions

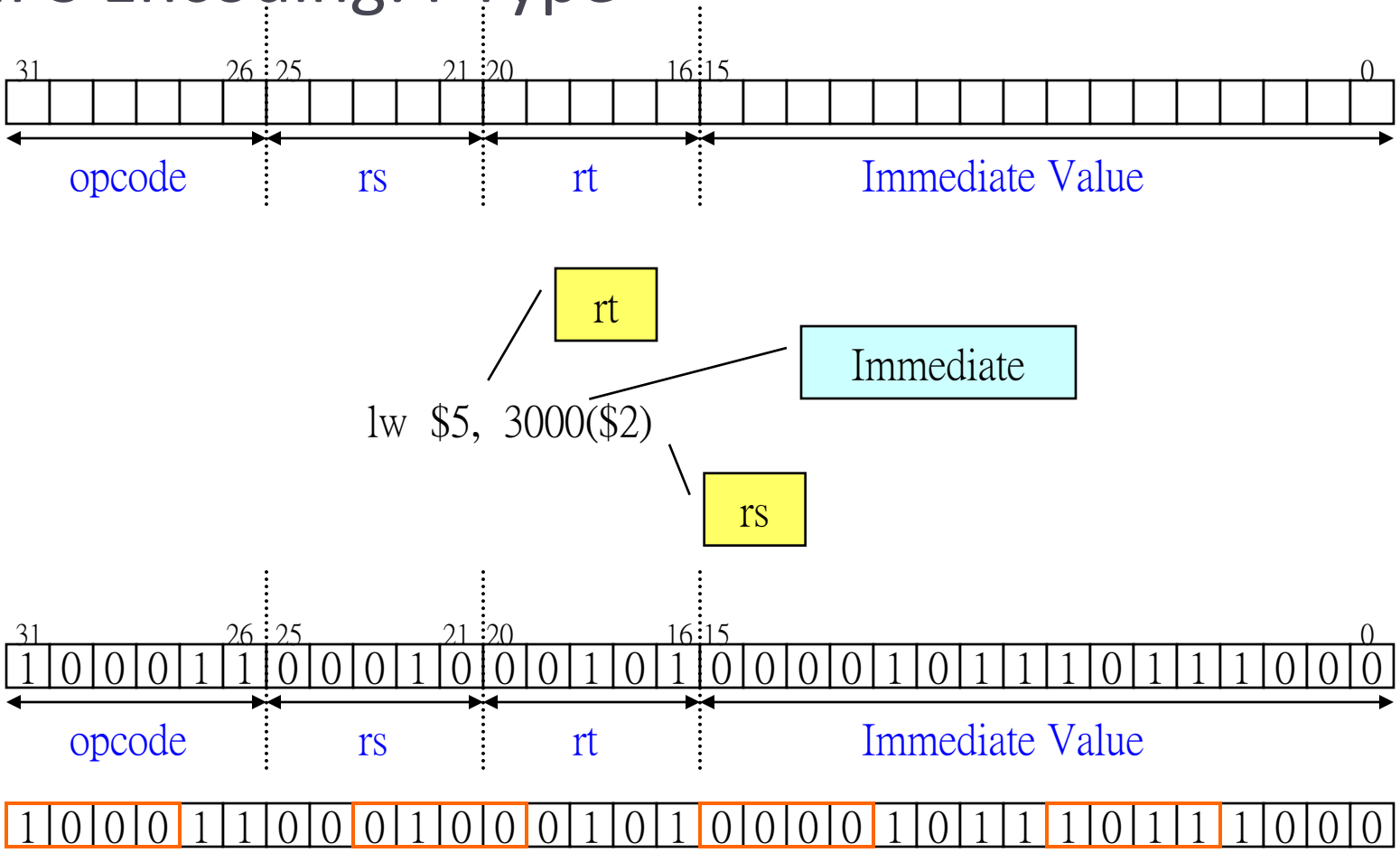
- Design Principle 4: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: lw $t0, 32($s2)

35	18	8	32

op	rs	rt	16 bit number

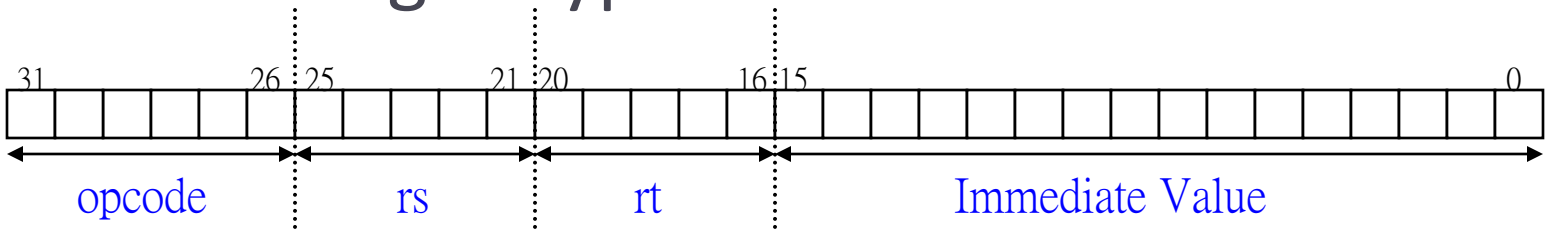
- Where's the compromise?

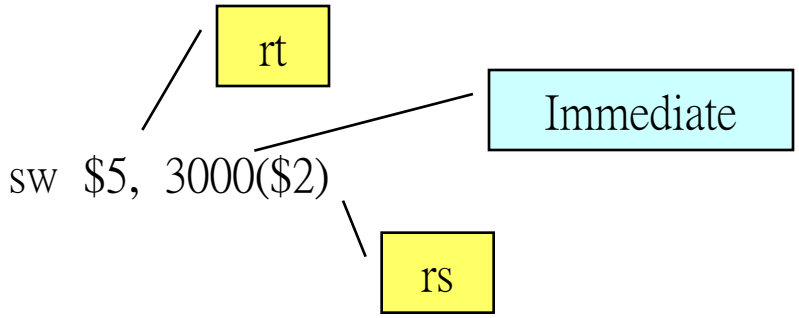
MIPS Encoding: I-Type

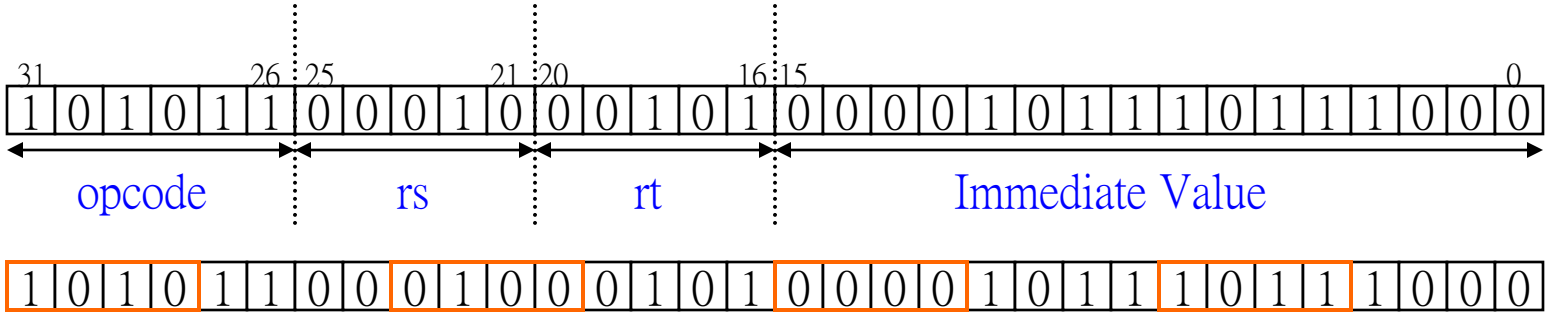


Encoding = 0x8C450BB8

MIPS Encoding: I-Type



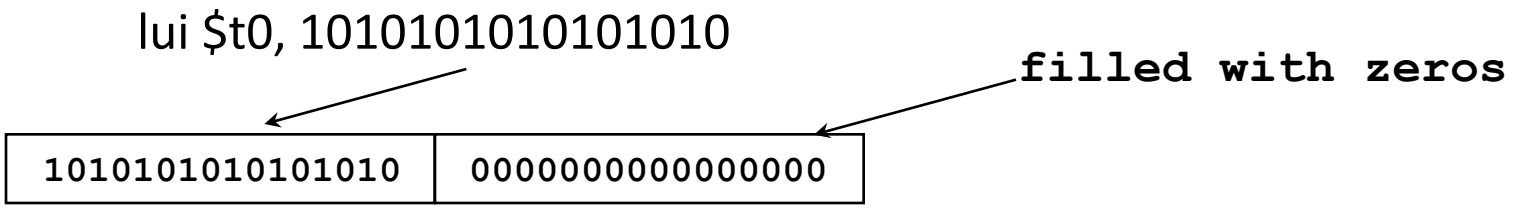




Encoding = 0xAC450BB8

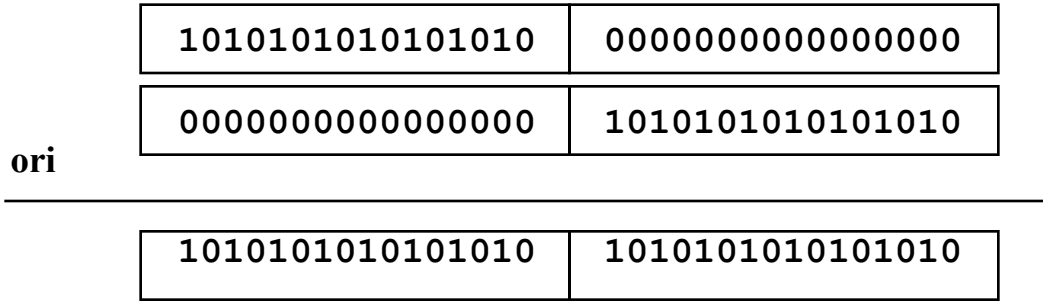
How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction



- Then must get the lower order bits right, i.e.,

```
addi $t0,$t0,1010101010101010
ori $t0, $t0, 1010101010101010
```

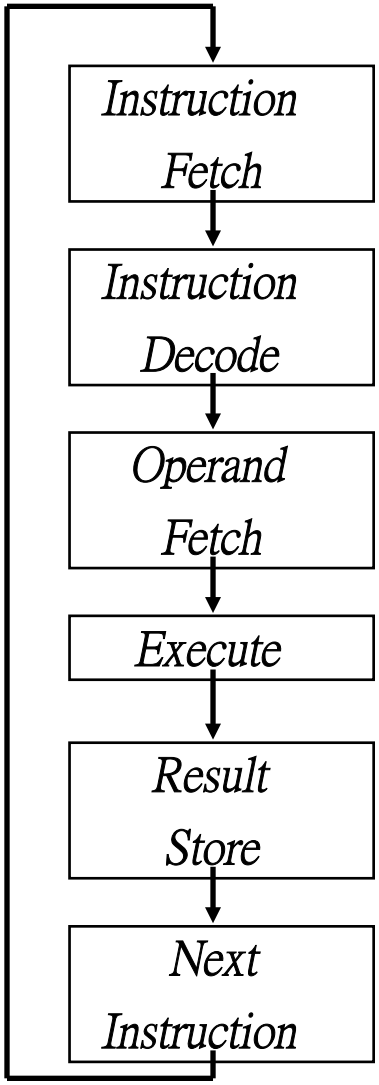


Stored Program Computers

- Von Neumann “invented” stored program computer in 1945
- Two key principles for machine design:

1. Instructions are represented as numbers and, as such, are indistinguishable from data
2. Programs are stored in alterable memory (that can be read or written to) just like data

Stored Program Concept



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

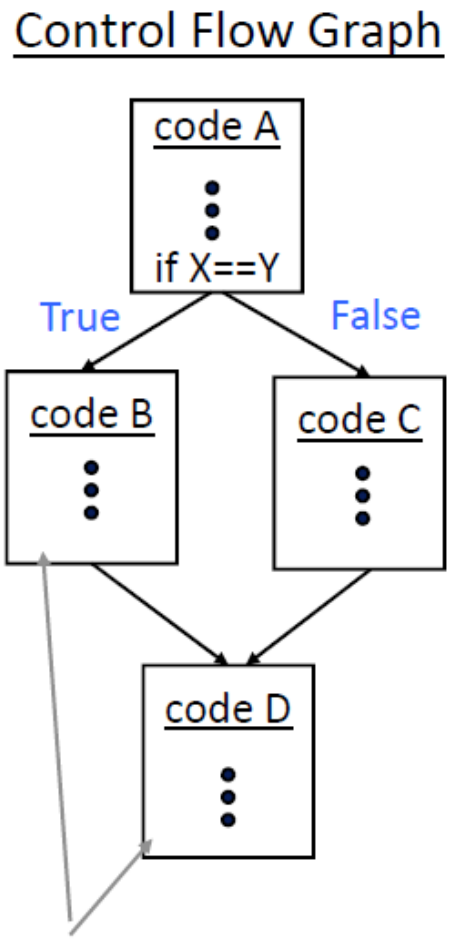
Deposit results in storage for later use

Determine successor instruction

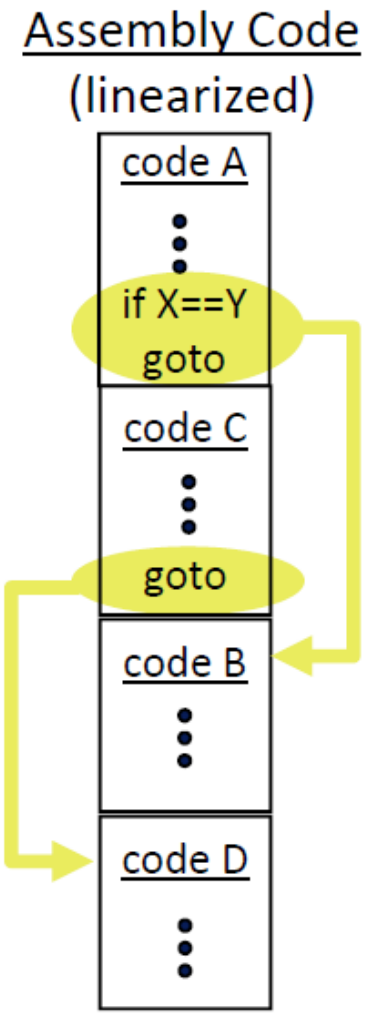
Control Flow Instructions

- C-Code

```
{ code A }
if X==Y then
    { code B }
else
    { code C }
{ code D }
```



these things are called basic blocks



Control Flow Instructions

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:
- **beq $t0, $t1, Label**
  - Go to the statement labeled Label if the value in $t0 equals the value in $t1
- **bne $t0, $t1, Label**
  - Go to the statement labeled Label if the value in $t0 *does not* equals the value in $t1

Control Flow Instructions

- Example: if (i==j) h = i + j; // I,j,h mapped to $s0-$s2

```
        bne $s0, $s1, Label
        add $s2, $s0, $s1
Label:  ....
```

Control Flow Instructions

- MIPS unconditional branch instructions:

- **j L1**
  - Unconditional jump
- **jr $t0**
  - “jump register” . Jump to the instruction specified in register $t0

if-then-else

- Example:

```
if ( a==b) x = y + z;          // x,y,z,a,b mapped to $s0-$s4
else x = y – z ;
```

```
    bne   $s3, $s4, Else        #  goto Else if a!=b
    add   $s0, $s1, $s2         # x = y + z
    j     Exit                  # goto Exit
Else : sub  $s0,$s1,$s2         # x = y – z
Exit :
```

Compiling Loop Statements

- C code:

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

  - i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:

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

Compiling Loop Statements

- C code:

```
    while ( A[i] == k )          // i,j,k in $s3. $s4, $s5
            i =  i + j;                  // A is in $s6

Loop: sll $t1, $s3, 2        # $t1 = 4 * i
      add  $t1, $t1, $s6     # $t1 = addr. Of A[i]
      lw   $t0, 0($t1)       # $t0 = A[i]
      bne  $t0, $s5, Exit    # goto Exit if A[i]!=k
      add  $s3, $s3, $s4     # i = i + j
      j    Loop              # goto Loop
Exit:
```

Control Flow Instructions

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
slt $t0, $s1, $s2
```

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

- Branch less than
  - Example: `if(A < B) goto LESS`

```
slt     $t1, $s1, $s2     #t1 = 1 if A < B
bne     $t1, $0, LESS
```

Signed vs. Unsigned

- Signed comparison: slt , slti
- Unsigned comparison: sltu , sltui
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - slt $t0, $s0, $s1 # signed
    - –1 < +1 ⇒ $t0 = 1
  - sltu $t0, $s0, $s1 # unsigned
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Switch Statement

- C code:

```
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;
        }
```

f-k in $s0-$s5 and $t2 contains 4 (maximum of var k)

- The switch statement can be converted into a big chain of if-then-else statements.
- A more efficient method is to use a jump address table of addresses of alternative instruction sequences and the **jr** instruction. Assume the table base address in $t4

Switch Statement

```
        slt     $t3, $s5, $zero     # is k < 0
        bne     $t3, $zero, Exit    # if k < 0, goto Exit
        slt     $t3, $s5, $t2       # is k < 4, here $t2=4
        beq     $t3, $zero, Exit    # if k >=4 goto Exit
        sll     $t1, $s5, 2         # $t1 = 4 * k
        add     $t1, $t1, $t4       # $t1 = addr. Of $t4[k]
        lw      $t0, 0($t1)         # $t0  = $t4[k]
        jr      $t0                 # jump to addr. In $t0

# $t4[0]=&L0, $t4[1]=&L1, …,
L0 :    add     $s0, $s3, $s4       # f = i + j
        j       Exit
L1 :    add     $s0, $s1, $s2       # f = g + h
        j       Exit
L2 :    sub     $s0, $s1, $s2       # f = g – h
        j       Exit
L3 :    sub     $s0, $s1, $s2       # f = i – j
Exit :
```

Procedure calls

- Procedures or subroutines:
  - Needed for structured programming
  - Reuse of code
- Steps followed in executing a procedure call:
  - Place parameters in a place where the procedure (callee) can access them
  - Transfer control to the procedure
  - Acquire the storage resources needed for the procedure
  - Perform desired task
  - Place results in a place where the calling program (caller) can access them
  - Return control to the point of origin

Resources Involved

- Registers used for procedure calling:
  - $a0 - $a3 : four argument registers in which to pass parameters
  - $v0 - $v1 : two value registers in which to return values
  - $ra : one return address register to return to the point of origin
- C code:

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

  - Arguments g, …, j are passed in $a0, …, $a3
  - f in $s0 (we need to save $s0 on stack)
  - Results are returned in $v0, $v1

Procedure Call Instructions

- Procedure call: jump and link

  jal ProcedureAddress

- Address of following instruction (PC+4) put in $ra
- Jumps to target address
- Transferring the control to the callee
- Procedure return: jump register

  jr $ra

- Copies $ra to program counter
- Returning the control to the caller
- Can also be used for computed jumps
  - e.g., for case/switch statements

Where is all this stuff saved to?

- Stack
  - A dedicated area of memory
- Stack operations
  - push: place data on stack (sw in MIPS)
  - pop: remove data from stack (lw in MIPS)
- Stack pointer
  - Stores the address of the top of the stack
  - $29 ($sp) in MIPS

Registers and Parameters

- The registers must be considered as global memory locations among the different subroutines.
- Someone needs to insure after the subroutine returns, that registers contain the old values that they had before it was called
- Multiple possible approaches:
  - Before every subroutine call, the caller saves all the registers that it will need (regardless of the ones used by the callee), and restores them after the subroutine returns, or
  - The callee saves the registers that it will use in its body, and restores all of them (regardless of the ones used by its caller).

Registers and Parameters

- MIPS:
- A compromise: Divide registers between those saved by caller (t registers) and those saved by callee (s registers).

Example

- MIPS code:

```
leaf_example:
    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
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

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

Non-Leaf Procedure Example

- C code:

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

  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

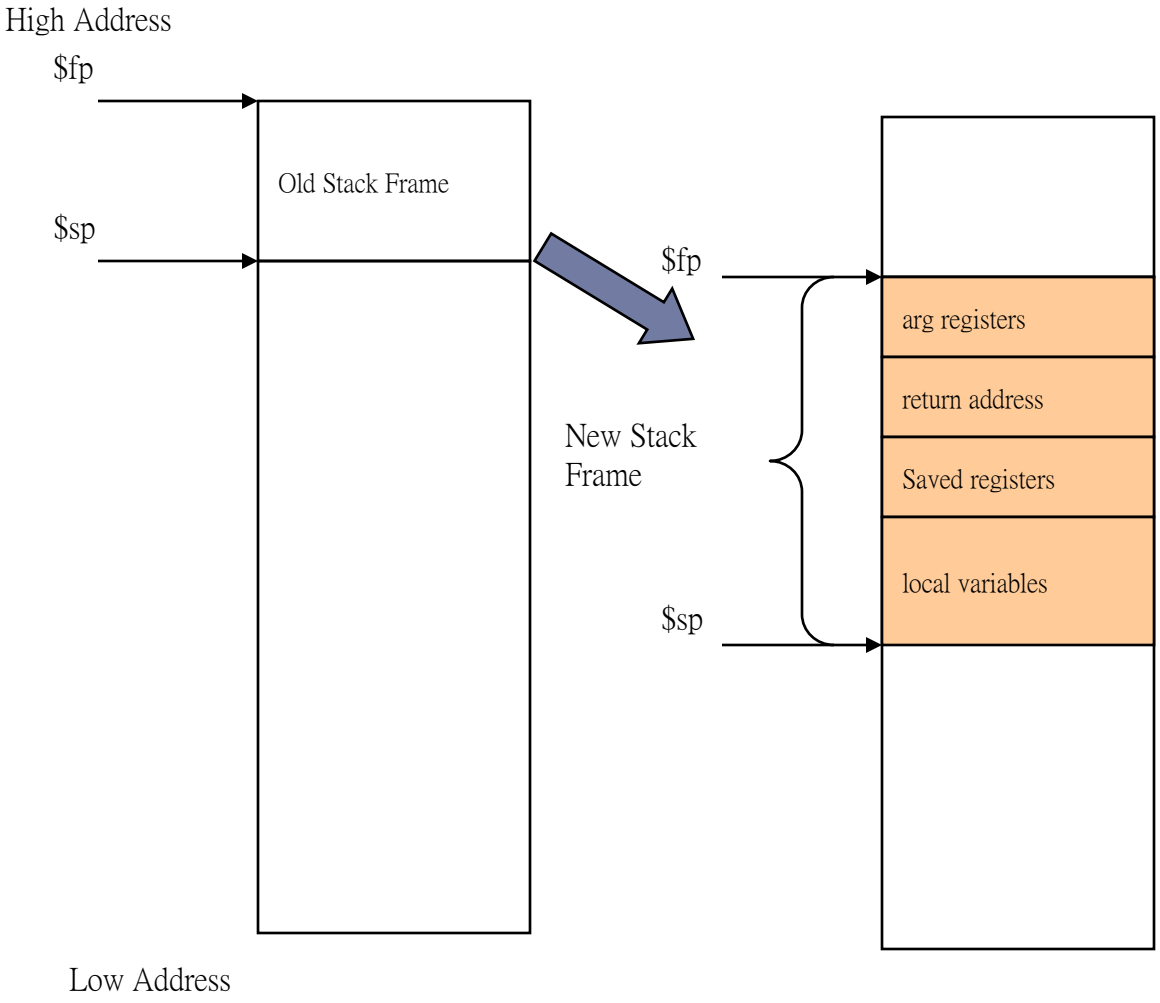
- MIPS code:

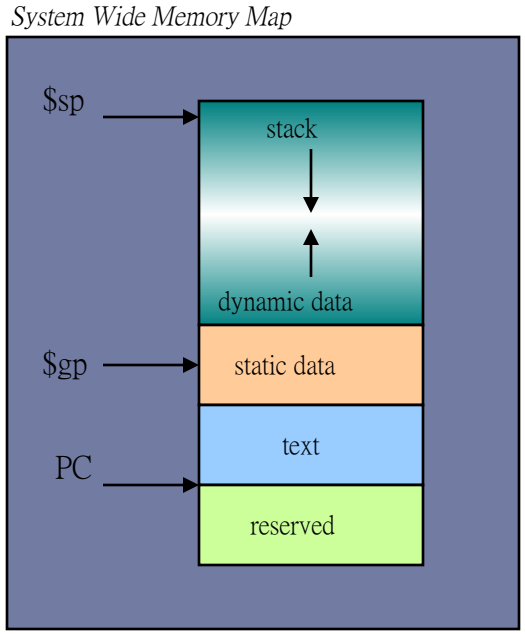
```
fact:
    addi $sp, $sp, -8       # adjust stack for 2 items
    sw   $ra, 4($sp)        # save return address
    sw   $a0, 0($sp)        # save argument
    slti $t0, $a0, 1        # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        #   pop 2 items from stack
    jr   $ra                #   and return
L1: addi $a0, $a0, -1       # else decrement n
    jal  fact               # recursive call
    lw   $a0, 0($sp)        # restore original n
    lw   $ra, 4($sp)        #   and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul  $v0, $a0, $v0      # multiply to get result
    jr   $ra                # and return
```

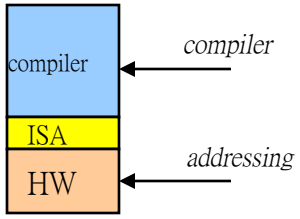
Procedure Frame/Stack Frame/Activation Record

- Each procedure is associated with a call frame
- Each frame has a frame pointer: $fp

Procedure Frame/Stack Frame/Activation Record







Procedure Call: Summary

- Done by the Caller
  - Save the t registers used by the caller
  - Save the arguments sent to subroutine
  - Store return address and jump to subroutine (jal)
  - Restore the t registers used by the caller
- Done by the Callee
  - Save the s registers used in the subroutine body
  - Save the return address ($ra), if necessary
  - Restore the s registers
  - Restore value of $ra, if necessary

Policy of Use Conventions: Register Usage

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$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

Compiling a String Copy Procedure

```
void      strcpy ( char x[ ], y[ ])
{         int i=0;
          while ( x[ i ] = y [ i ] != 0)
                    i ++ ;
}                   // x and y base addr. are in $a0 and $a1
strcpy :
          addi    $sp, $sp, -4          #  reserve 1 word space in stack
          sw      $s0, 0($sp)           # save $s0
          add     $s0, $zer0, $zer0     # i = 0
L1 :      add     $t1, $a1, $s0         # addr. of y[ i ] in $t1
          lb      $t2, 0($t1)           # $t2 = y[ i ]
          add     $t3, $a0, $s0         # addr. Of x[ i ] in $t3
          sb      $t2, 0($t3)           # x[ i ] = y [ i ]
          beq     $t2, $zero, L2        # if y [ i ] = 0 goto L2
          addi    $s0, $s0, 1           # i ++
          j       L1                    # go to L1
```

Compiling a String Copy Procedure

```
void    strcpy ( char x[ ], y[ ])
{       int i=0;
        while ( x[ i ] = y [ i ] != 0)
                i ++ ;
}               // x and y base addr. are in $a0 and $a1


L2 :    lw      $s0, 0($sp)             # restore $s0
        addi    $sp, $sp, 4             # restore $sp
        jr      $ra                     # return
```

Compiling a String Copy Procedure

- Java uses Unicode for characters
- Unicode uses 16 bits to represent a character
- MIPS Instruction
  - Load half (lh): lh $t0, 0($sp)
  - Store half (lh): sh $t0, 0($sp)

Addresses in Branches and Jumps

- Instructions:

  bne $t4,$t5,Label Next instruction is at Label if $t4 ≠ $t5

  beq $t4,$t5,Label Next instruction is at Label if $t4 = $t5

  j Label Next instruction is at Label

- Formats:

- Addresses are not 32 bits
  - How do we handle this with large programs?
  - First idea: limitation of branch space to the first 2^{16} bits

Addresses in Branches

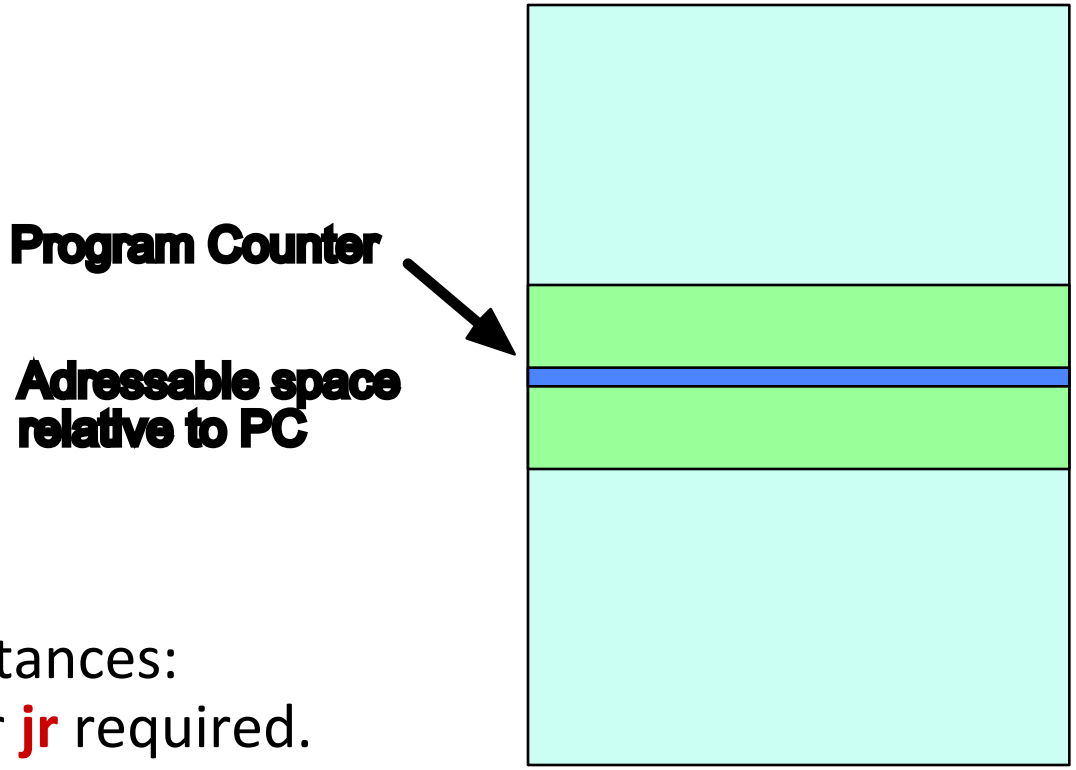
- Instructions:

  bne $t4,$t5,Label Next instruction is at Label if $t4≠$t5
  beq $t4,$t5,Label Next instruction is at Label if $t4=$t5

- Formats:

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)

PC-relative addressing

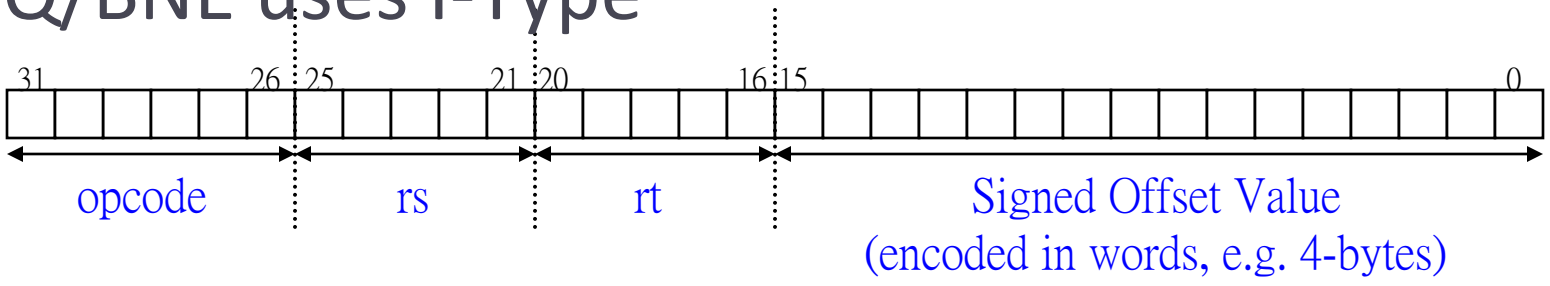


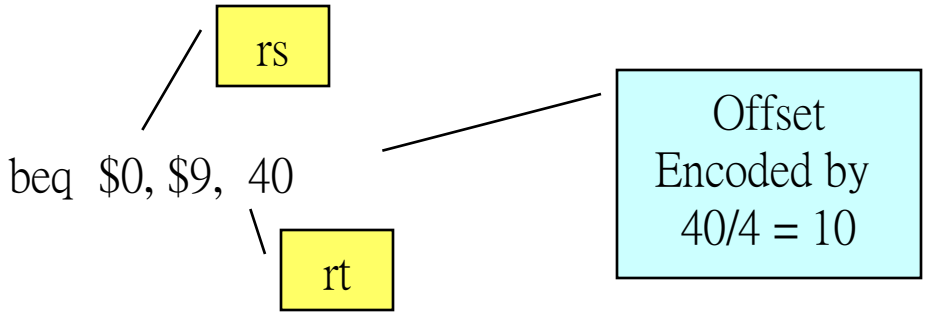
- For larger distances:
  Jump register **jr** required.

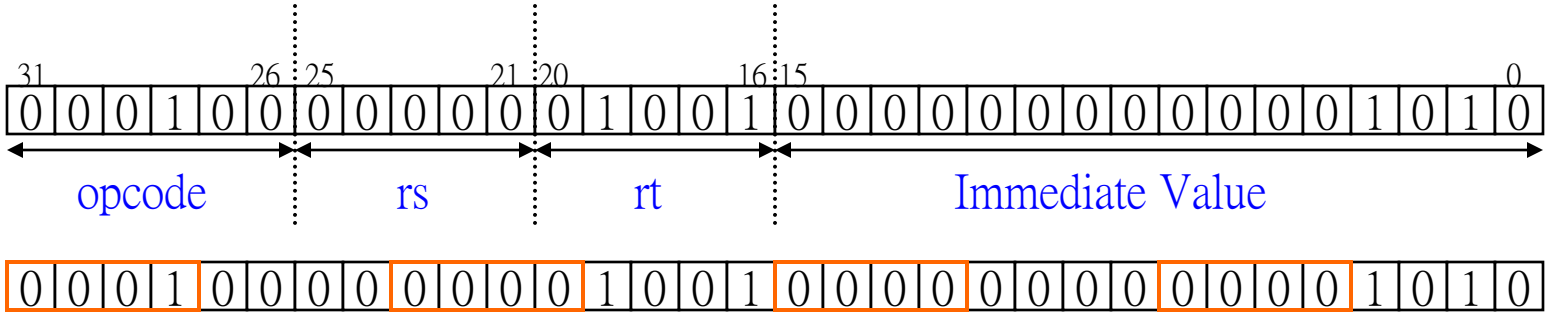
Addresses in Branches

- Jump instructions just use the high order bits of PC – Pseudodirect addressing
  - 32-bit jump address = 4 Most Significant bits of PC concatenated with 26-bit word address (or 28- bit byte address)
  - Address boundaries of 256 MB

BEQ/BNE uses I-Type

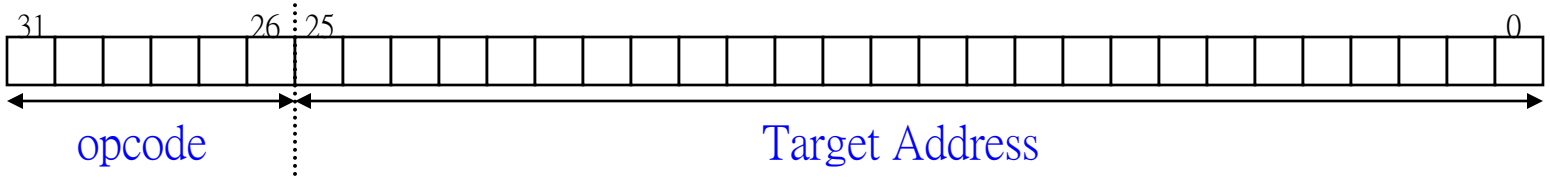






Encoding = 0x1009000A

MIPS Encoding: J-Type



ñjal will jump and push return address in $ra ($31)

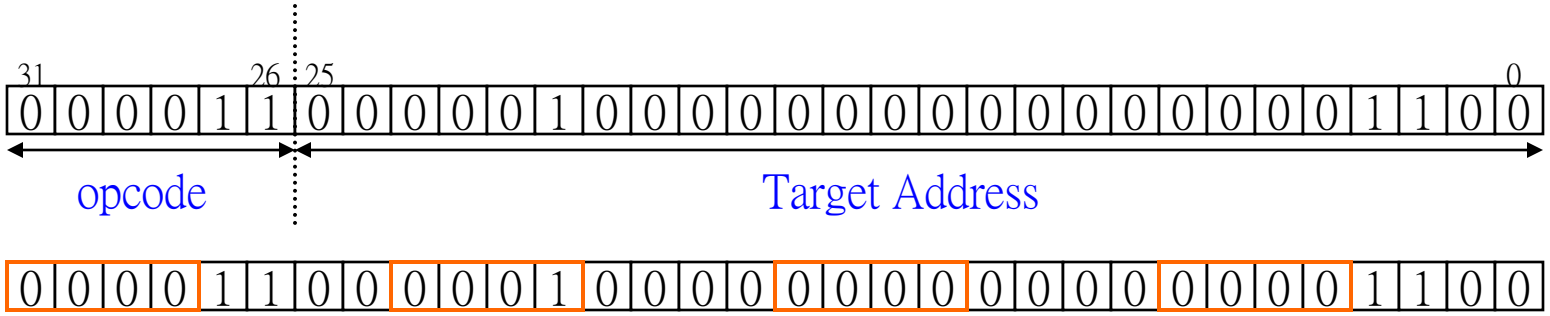
jal 0x00400030

Target

0000 0000 0100 0000 0000 0000 0011 0000 X

Target Address

Instruction=4 bytes

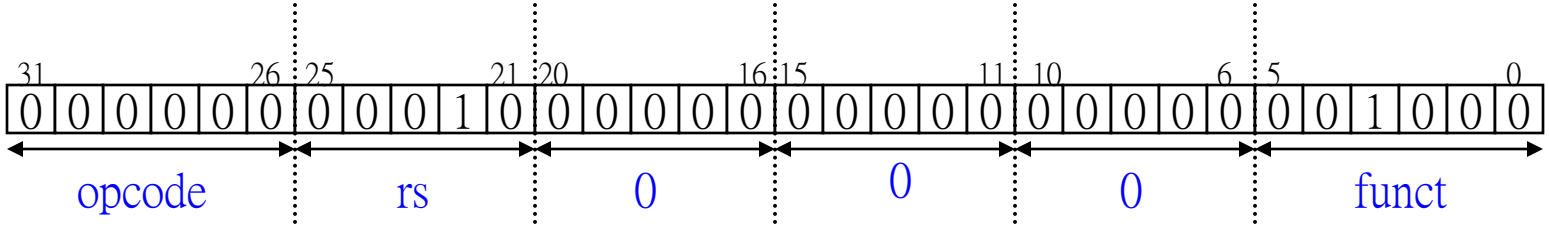


Encoding = 0x0C10000C

JR

- JR (Jump Register)
  - Unconditional jump

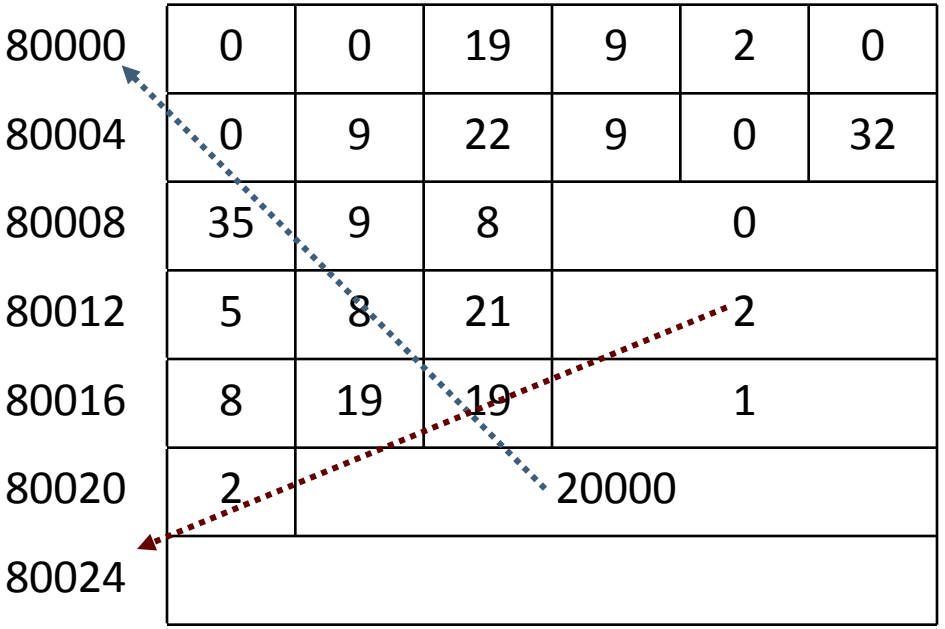
jr $2



Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

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



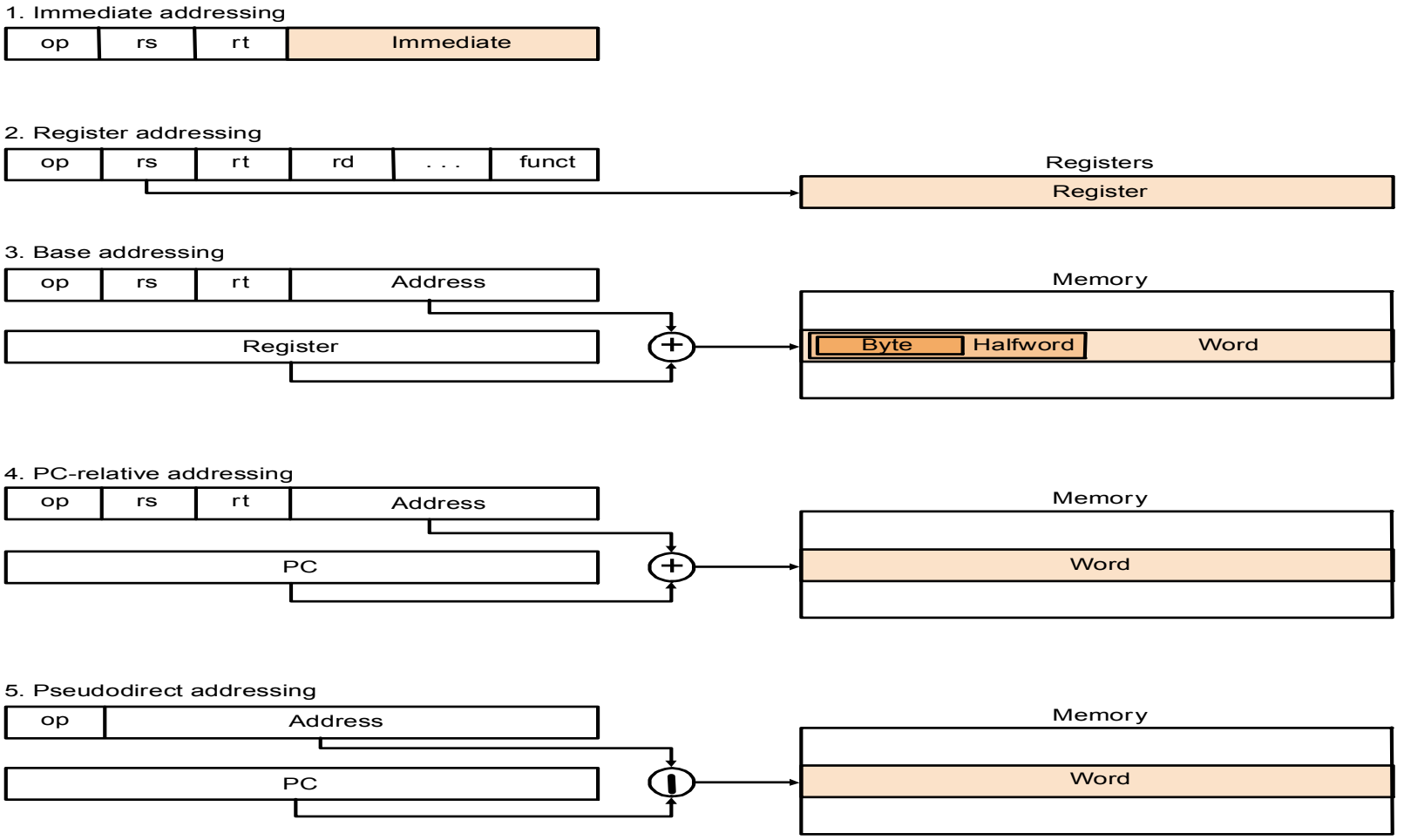
Branching Far Away

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

- Example

```
            beq $s0,$s1, L1
                    ↓
            bne $s0,$s1, L2
            j L1
L2:         …
```

MIPS Addressing Modes



Array vs. Pointers

```
Clear1 ( int array [ ] , int size)
{       int i ;
        for ( i = 0; i < size; i ++)
                array [ i ] = 0;
}

Clear2 ( int *array , int size)
{       int *p;
        for(p=&array[0]; p<&array[size]; p ++)
                *p = 0;
}
// $a0 = addr. of array, $a1 has size, $t0=i
```

Array vs. Pointers: : MIPS for Array Version

```
Clear1 ( int array [ ] , int size)
{       int i ;
        for ( i = 0; i < size; i ++)
                array [ i ] = 0;
}
```

// $a0 = addr. of array, $a1 has size, $t0 = i

```
        move  $t0, $zero          #  i = 0
Loop1:  sll   $t1, $t0, 2         #  $t1 = 4 * i
        add   $t2, $a0, $t1       #  $t2 = addr. of array
        sw    $zero, 0($t2)       #  array[i] = 0
        addi  $t0, $t0,1          #  i ++
        slt   $t3, $t0, $a1       # check end of loop (i<size)
        bne   $t3, $zero, Loop1   # if i < size goto Loop1
```

Array vs. Pointers: : MIPS for Pointer Version

```
Clear2 ( int *array , int size)
{       int *p;
        for(p=&array[0]; p<&array[size]; p ++)
                *p = 0;
}
```

// $a0 = addr. of array, $a1 has size, $t0 = p

```
        move   $t0, $a0          #  p = addr. of array[0]
        sll    $t1, $a1, 2       # $t1 = 4 * size
        add    $t2, $a0, $t1     #$t2 = addr. of array[size]
Loop2:  sw     $zero, 0($t0)     #  store 0 in *p
        addi   $t0, $t0, 4       #  p = p + 4
        slt    $t3, $t0, $t2     #  $t3=(p<&array[size]) #if
        bne    $t3, $zero, Loop2 p < last addr. Go Loop2
```

Array vs. Pointers: : MIPS for Pointer Version

- The pointer version reduces the # of instructions per iteration from 6 to 4
- Many optimizing compilers will generate this code, even for array-based C code

To summarize:

MIPS operands		
Name	**Example**	**Comments**
	`$s0-$s7, $t0-$t9, $zero,`	Fast locations for data. In MIPS, data must be in registers to perform
32 registers	`$a0-$a3, $v0-$v1, $gp,`	arithmetic. MIPS register $zero always equals 0. Register $at is
	`$fp, $sp, $ra, $at`	reserved for the assembler to handle large constants.
	Memory[0],	Accessed only by data transfer instructions. MIPS uses byte addresses, so
2^{30} memory	Memory[4], ...,	sequential words differ by 4. Memory holds data structures, such as arrays,
words	Memory[4294967292]	and spilled registers, such as those saved on procedure calls.

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
Conditional branch	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Summary: MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI

Intel IA-32

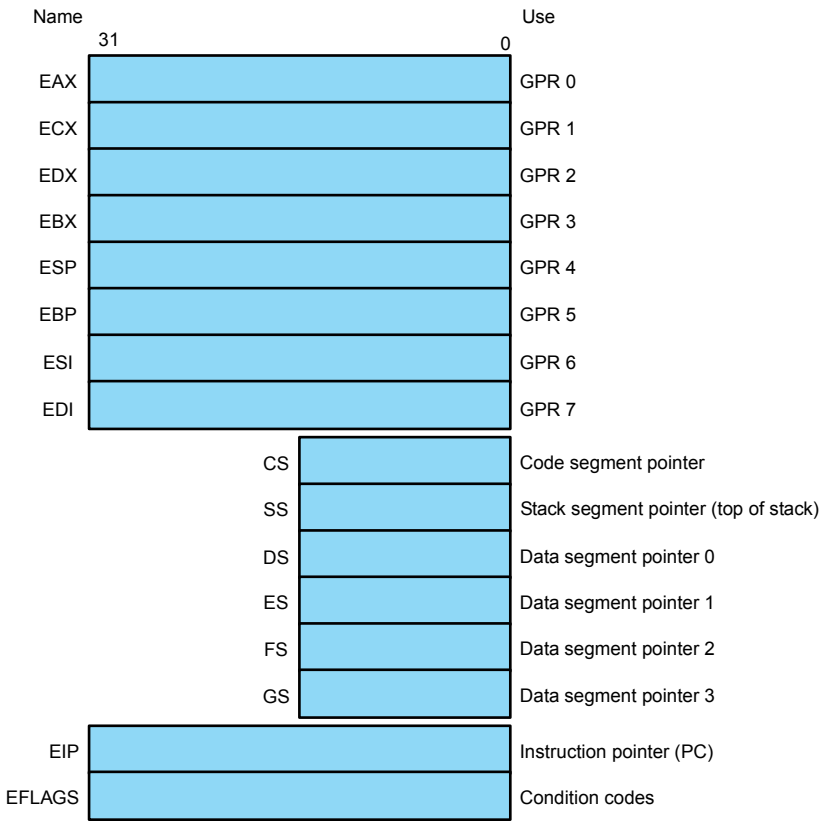
- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX (SIMD-INT) is added (PPMT and P-II)
- 1999: SSE (single prec. SIMD-FP and cacheability instructions) is added in P-III
- 2001: SSE2 (double prec. SIMD-FP) is added in P4
- 2004: Nocona introduced (compatible with AMD64 or once called x86-64)

IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

IA-32 Registers & Data Addressing

- Registers in the 32-bit subset that originated with 80386



IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1)` `#` ≤16-bit `#displacement`
Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,0($t0)`
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,100($t0)` `#` ≤16-bit `#displacement`

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

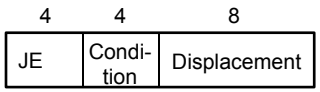
Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

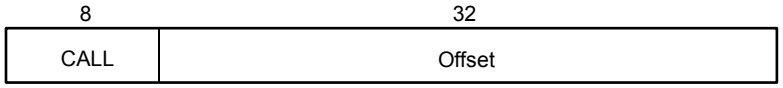
IA-32 instruction Formats

- Typical formats: (notice the different lengths)

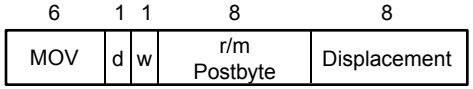
a. JE EIP + displacement



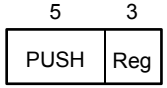
b. CALL



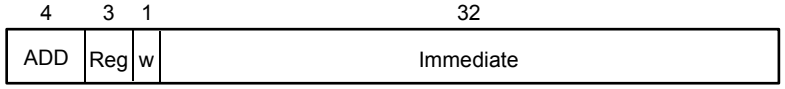
c. MOV EBX, [EDI + 45]



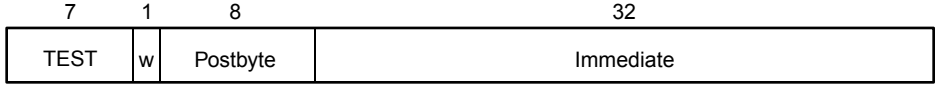
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



Reading Materials

- Class Lectures
- Computer Organization and Design (3rd Edition)
  - Section 2.1 – 2.9, 2.13-2.20
- Computer Architecture, A quantitative Approach
  - Section A.1-A.2

Acknowledgements

- These slides contain material developed and copyright by:
  - Krste Asanovic (UCB) (Course# 152, Spring 2012)
  - James Hoe (CMU) (Course# 18-447, Spring 2011)
  - Li-Shiuan Peh (MIT) (Course# 6.823, Fall 2012)
  - Sudhakar Yalamanchili (GATECH) (Course# ECE3056, Fall 2012)
  - Amirali Baniasadi (UVIC) (Course# CENG450, Summer 2012)
  - Xiaoyu Zhang (CSUSM) (Course# CS331, Spring 2008)

The End