# Lecture 4

- **Instruction Set Architecture (II)**

# Procedures

- int f1 (int i, int j, int k, int g)

```
{ ::::
 add $3, $4, 1;
  return 1;
 }
```

**callee**

```
 int f2 (int s1, int s2)
 {
   ::::::
  add   $3, $4, $3
   i = f1 (3,4,5, 6);
   add   $2, $3, $3
   ::::
 }
```

**caller**

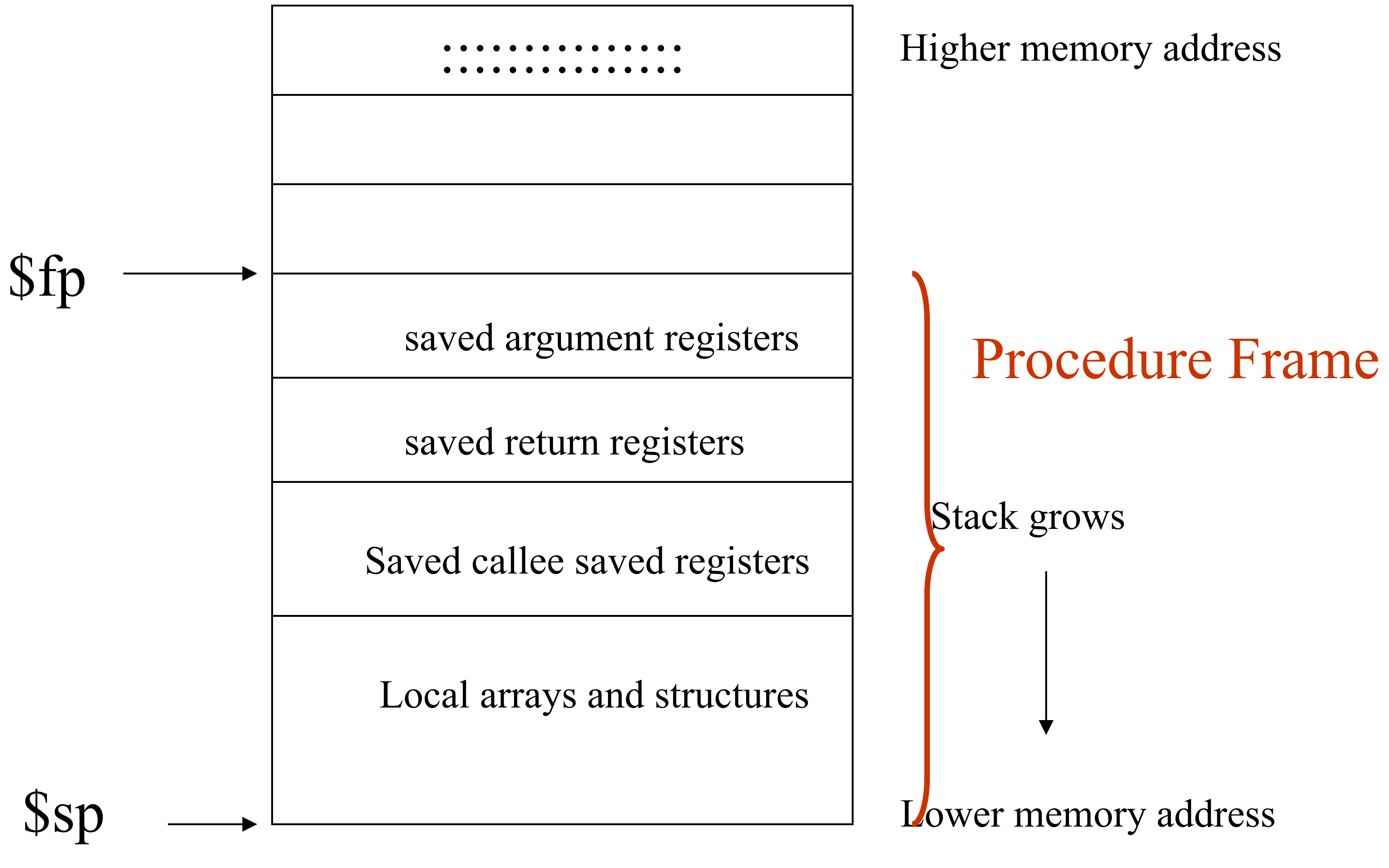
- How to pass parameters & results?
- How to preserve caller register values?
- How to alter control? (I.e. go to callee, return from callee)

# Procedures

- How to pass parameters & results
  - $a0-$a3: four argument registers. What if # of parameters is larger than 4? – push to the stack
  - $v0-$v1: two value registers in which to return values
- How to preserve caller register values?
  - Use stack
  - Caller saved register
  - Callee saved register
- How to switch control?
  - How to go to the callee
    - Jal procedure_address  (jump and link)
      - Store the the return address  (PC +4 ) at $ra
      - set PC  = procedure_addres
  - How to return from the callee
    - Callee exectues **jr $ra**

```
int f1 (int i, int j, int k, int g)
  { ::::
   add $3, $4, 1;
    return 1;
   }

int f2 (int s1, int s2)
   {
     ::::::
    add   $3, $4, $3
PC →  i = f1 (3,4,5, 6);
     add   $2, $3, $3
     ::::
   }
```

# Procedure Call Stack (Frame)



::::::::::::::::

Higher memory address

$fp

saved argument registers

Procedure Frame

saved return registers

Stack grows

Saved callee saved registers

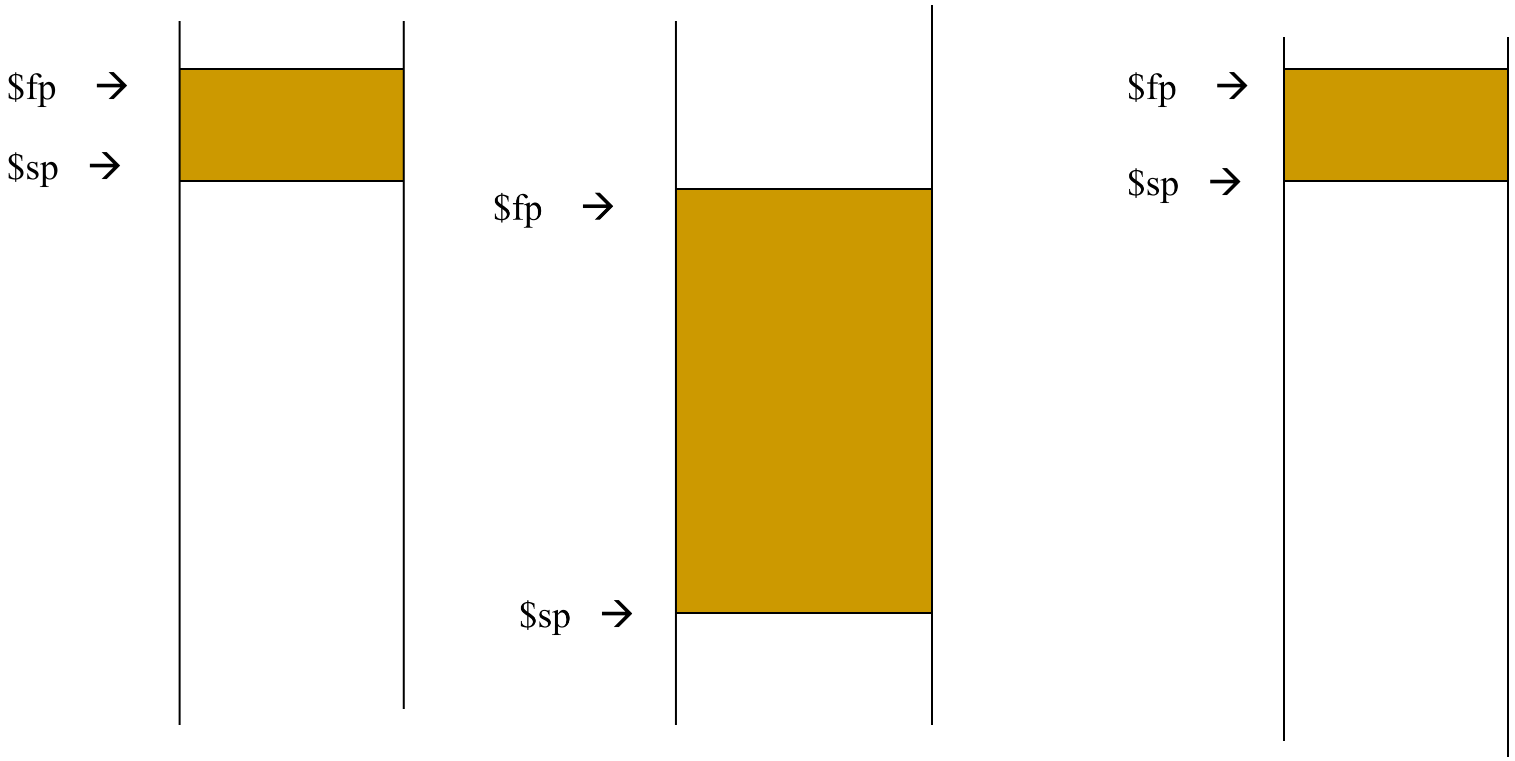
Local arrays and structures

$sp

Lower memory address

- Frame pointer points to the first word of the procedure frame

# Procedure Call Stack (Frame)



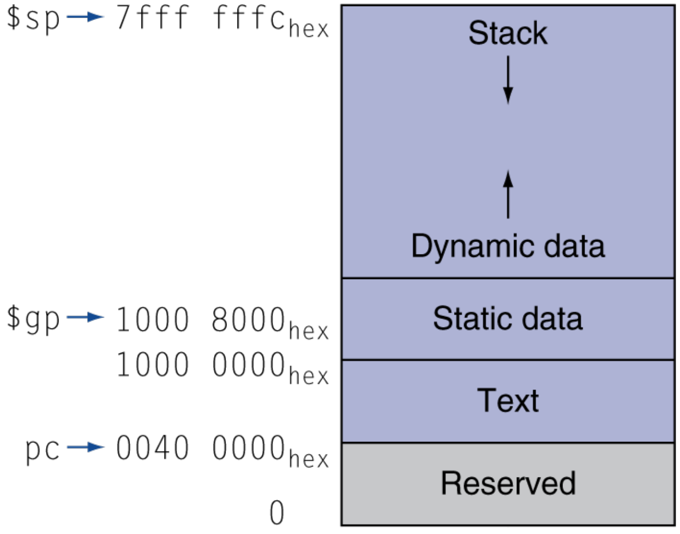
Before the procedure call

during the procedure call

after the procedure call

# Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



# Procedure Calling Convention

- Calling Procedure
  - Step-1: pass the argument
  - Step-2: save caller-saved registers
  - Step-3: Execute a jal instruction

```
foo1 ()
{   ::::::::
    i= i+1;  t3
    x=foo(4);
    i = x + i
}
```

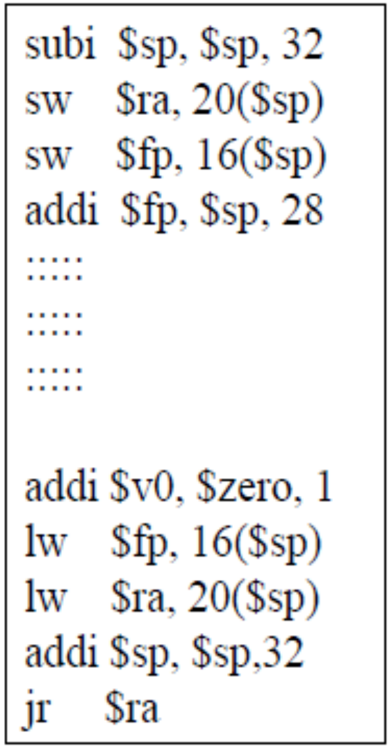
```
  ::::::::
li    $a0, 4          # passing argument
sw    $t3, 4($sp)    # save $t3
jal   foo
lw    $t3, 4($sp)  #  restore $t3
add   $t3, $v0, $t3

   ::::::
```

# Procedure Calling Convention

Called Procedure

- Step-1: establish stack frame
  - subi $sp, $sp <frame-size>
- Step-2: saved callee saved registers
  - $ra, $fp,$s0-$s7
- Step-3: establish frame pointer
  - Add $fp, $sp, <frame-size>-4

- On return from a call
  - Step-1: put returned values in
  - register $v0, $v1.
  - Step-2: restore callee-saved registers
  - Step-3: pop the stack
  - Step-4: return: jr $ra



```
subi  $sp, $sp, 32
sw    $ra, 20($sp)
sw    $fp, 16($sp)
addi  $fp, $sp, 28
:::::
:::::
:::::

addi $v0, $zero, 1
lw    $fp, 16($sp)
lw    $ra, 20($sp)
addi $sp, $sp,32
jr     $ra
```

# Preserved Registers

| | | |
|---|---|---|
| 0 | zero | constant 0 |
| 1 | at | reserved for assembler |
| 2 | v0 | expression evaluation & |
| 3 | v1 | function results |
| 4 | a0 | arguments |
| 5 | a1 | |
| 6 | a2 | |
| 7 | a3 | |
| 8 | t0 | temporary: caller saves |
| | . . . | |
| 15 | t7 | |
| 16 | s0 | callee saves |
| | . . . | |
| 23 | s7 | |
| 24 | t8 | temporary (cont’d) |
| 25 | t9 | |
| 26 | k0 | reserved for OS kernel |
| 27 | k1 | |
| 28 | gp | Pointer to global area |
| 29 | sp | Stack pointer |
| 30 | fp | frame pointer |
| 31 | ra | Return Address (HW) |

# Nested Procedures

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

```
fact:
     addi   $sp, $sp, -8
     sw     $ra, 4($sp)    # save $ra
     slti   $t0, $a0, 1    # n< 1?
     beq    $t0, $zero, L1
     addi   $v0,$zero,1   #  return 1
     addi   $sp, $sp, 8   #  fix up the stack pointer & return
     jr     $ra
L1:  sw     $a0, 0($sp)   # save argument $a0
     addi   $a0,$a0,-1    # n = n-1
     jal    fact          # jal(n-1)
     lw     $a0, 0($sp)   # restore argument $a0
     mul    $v0, $a0, $v0 # return n x fact(n-1)
     lw     $ra, 4($sp)   # restore $ra
     addi   $sp, $sp, 8   # restore stack pointer
     jr     $ra           # return to the caller
```

# Representation of Characters

- ASCII (American Standard Code for Information Interchange)
  - Uses 8 bits to represent a character
  - MIPS provides instructions to move bytes:

    ```
    lb   $t0, 0($sp)          #Read byte from source
    sb   $t0, 0($gp)          #Write byte to destination
    ```
- Unicode (Universal Encoding)
  - Uses 16 bits to represent a character
  - Used in Java
  - MIPS provides instructions to move 16 bits:

    ```
    lh   $t0, 0($sp)          #Read halfword from source
    sh   $t0, 0($gp)          #Write halfword to destination
    ```

# Array vs. Pointer

```
Clear1(int array[ ], int size)
{
    int i;
    for (i=0, i< size; i+= 1)
         array[i] = 0;
}
```

```
            move    $t0, $zero      # i =0
Loop1 : sll     $t1, $t0, 2       # I * 4
            add     $t2, $a0, $t1   # t2 = address of array[i]
            sw      $zero, 0($t2)   # array [i] = 0
            addi    $t0, $t0, 1       # i = i +1
            slt     $t3, $t0, $a1   # compare i and size
            bne     $t3,  $zero, loop1
```

# Array vs. Pointer (cont.)

```
Clear 2(int *array, int size)
{
    int *p,
    for (p = &array[0]; p < &array[size]; p = p+1)
        *p = 0;
}
```

```
        move    $t0, $a0          # p = &array[0]
        sll     $t1, $a1, 2       # t1 = size x 4
        add     $t2, $a0, $t1     # t2  = &array[size]
Loop2:  sw      $zero, 0($t0)     # memory[p] = 0
        addi    $t0, $t0, 4       #  p= p+4
        slt     $t3, $t0, $t2     #  compare p, & array[size]
        bne     $t3, $zero, Loop2
```

# Array vs. Pointer (cont.)

Array

```
         move    $t0, $zero      # i =0
Loop1 : sll      $t1, $t0, 2     # I * 4
         add     $t2, $a0, $t1   # t2 = address of array[i]
         sw      $zero, 0($t2)   # array [i] = 0
         addi    $t0, $t0, 1      # i = i +1
         slt     $t3, $t0, $a1    # compare i and size
         bne     $t3,  $zero, loop1
```

# of Instruction per iteration
 = 6

Pointer

```
         move    $t0, $a0              # p = &array[0]
         sll     $t1, $a1, 2           # t1 = size x 4
         add     $t2, $a0, $t1         # t2  = &array[size]
Loop2: sw        $zero, 0($t0)         # memory[p] = 0
         addi    $t0, $t0, 4           #  p= p+4
         slt     $t3, $t0, $t2         #  compare p, & array[size]
         bne     $t3, $zero, Loop2
```

# of Instruction per iteration
 = 4

# Parallelism and Instructions: Synchronization

- Parallel tasks must synchronize to avoid data race, where the results of the program can change depending on how events happen to occur.
- Lock/unlock: ensure only one task entering the critical section

```
P(1)
Acquire Lock;
If Lock = 0
   enter critical section;
Release Lock;
```

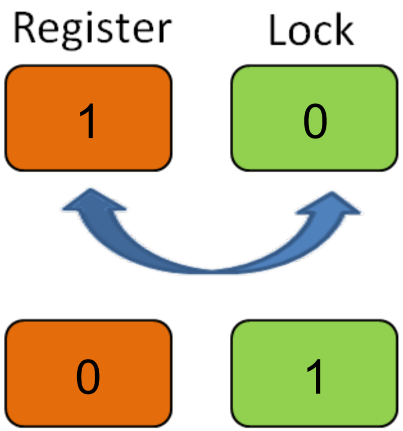
```
P(2)
Acquire Lock;
If Lock = 0
   enter critical section;
Release Lock;
```

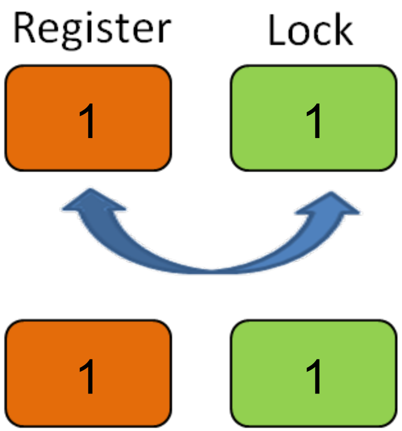
# Parallelism and Instructions: Synchronization

- **Atomic SWAP**: **atomically** **interchange a value in a register for a value in memory; nothing else can interpose itself between the read and the write to the memory location**

```
P(1)
$S4 =1;
Swap ($S4, Lock);
If Lock = 0
   enter critical section;
```

```
P(2)
$S4 =1;
Swap ($S4, Lock);
If Lock = 0
   enter critical section;
```





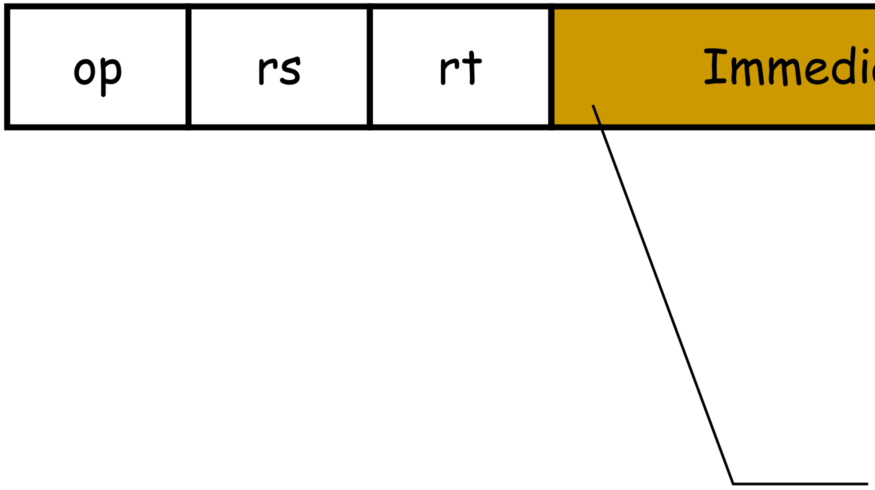
# Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

# MIPS Addressing Mode (1)

- Immediate addressing



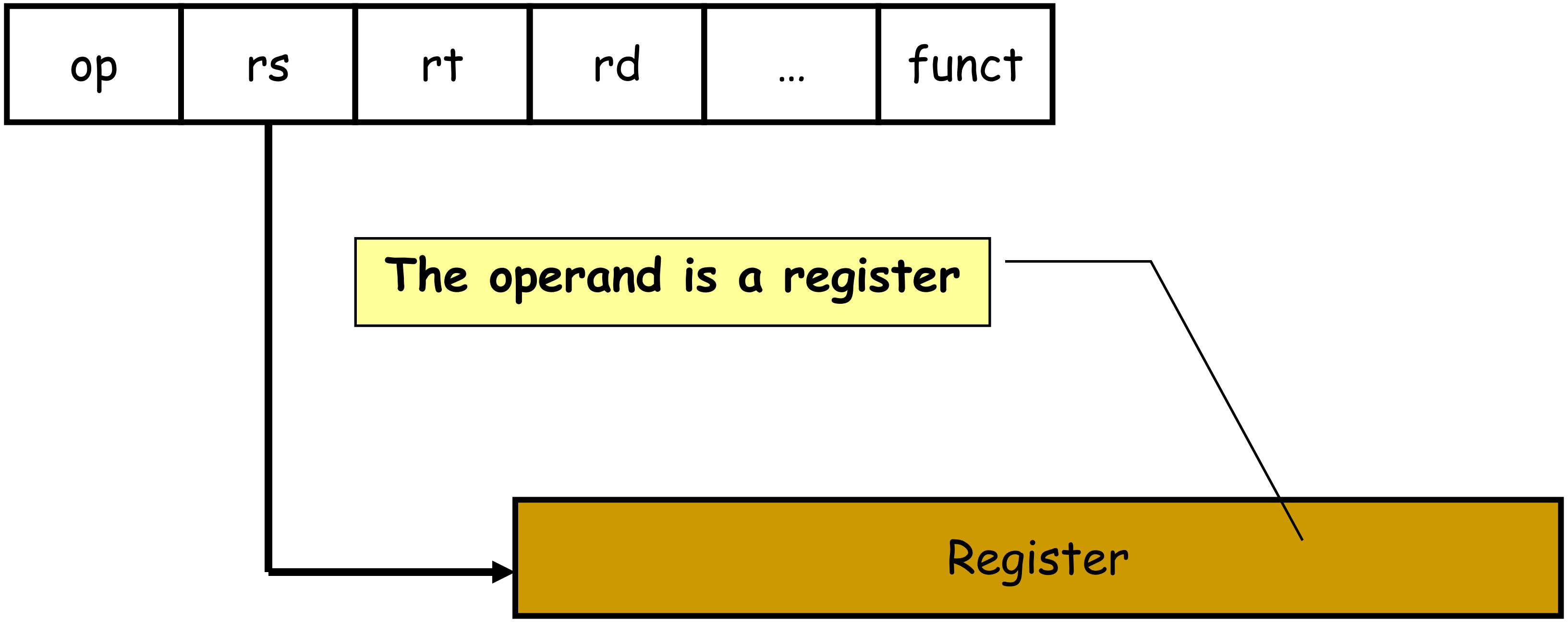
op | rs | rt | Immediate

The operand is a constant within the instruction itself

Example: addi $2, $3, 4

# MIPS Addressing Mode (2)

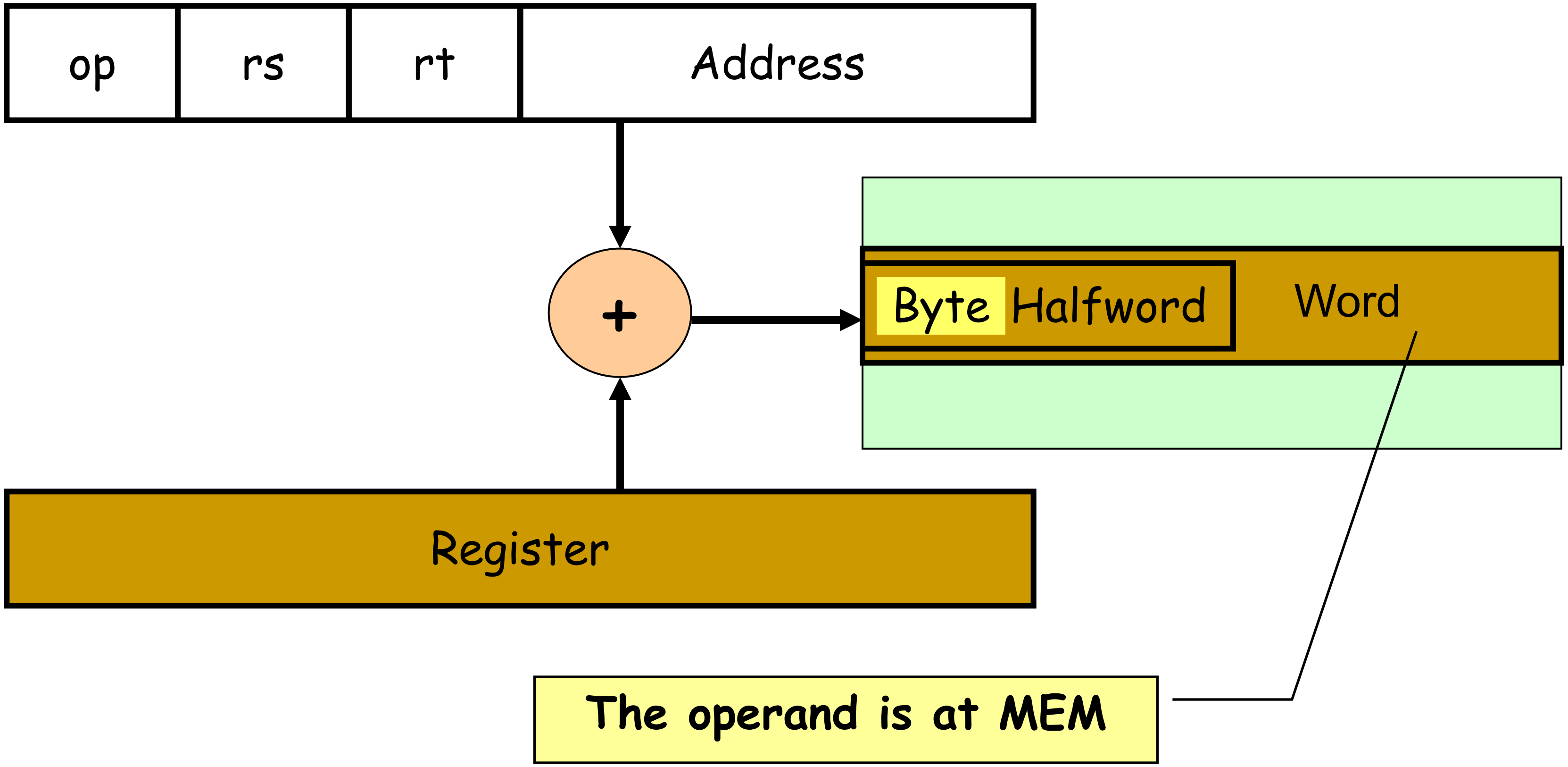
- Register addressing



Example : add $r1, $r2, $r3

# MIPS Addressing Mode (3)

- Base addressing



Example : lw $2, 100($3)

# How to Get the Base Address in the Base Register

**Method 1.**

```
        .data          # define prog. data section
xyz:   .word  1        # some data here
         …             # possibly some other data
        .text          # define the program code
         …             # lines of code
        lw   $5,xyz    # loads contents of xyz in r5
```

- the assembler generates an instruction of the form:
  lw   $5, offset($gp) # gp is register 28, the global pointer

Note : .data, .word, .text are assembler directives

# How to Get the Base Address in the Base Register (cont.)

**Method 2.**

```
la    $6,xyz     #r6 contains the address of xyz
lw    $5,0($6)   #r5 contains the contents of xyz
```

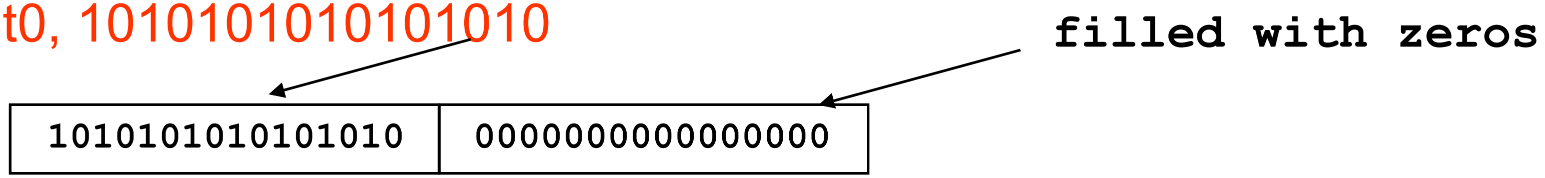
**Method 3. If the address is a constant**

li, *if less than ±32K*

lui and ori, otherwise.

# 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
- Example: load **`10101010101010101010101010101010 into register $t0`**
- lui $t0, 1010101010101010

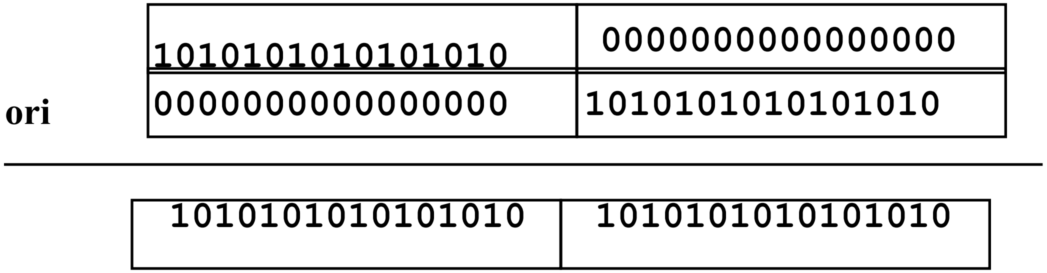


**`filled with zeros`**

| | |
|---|---|
| 1010101010101010 | 0000000000000000 |

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

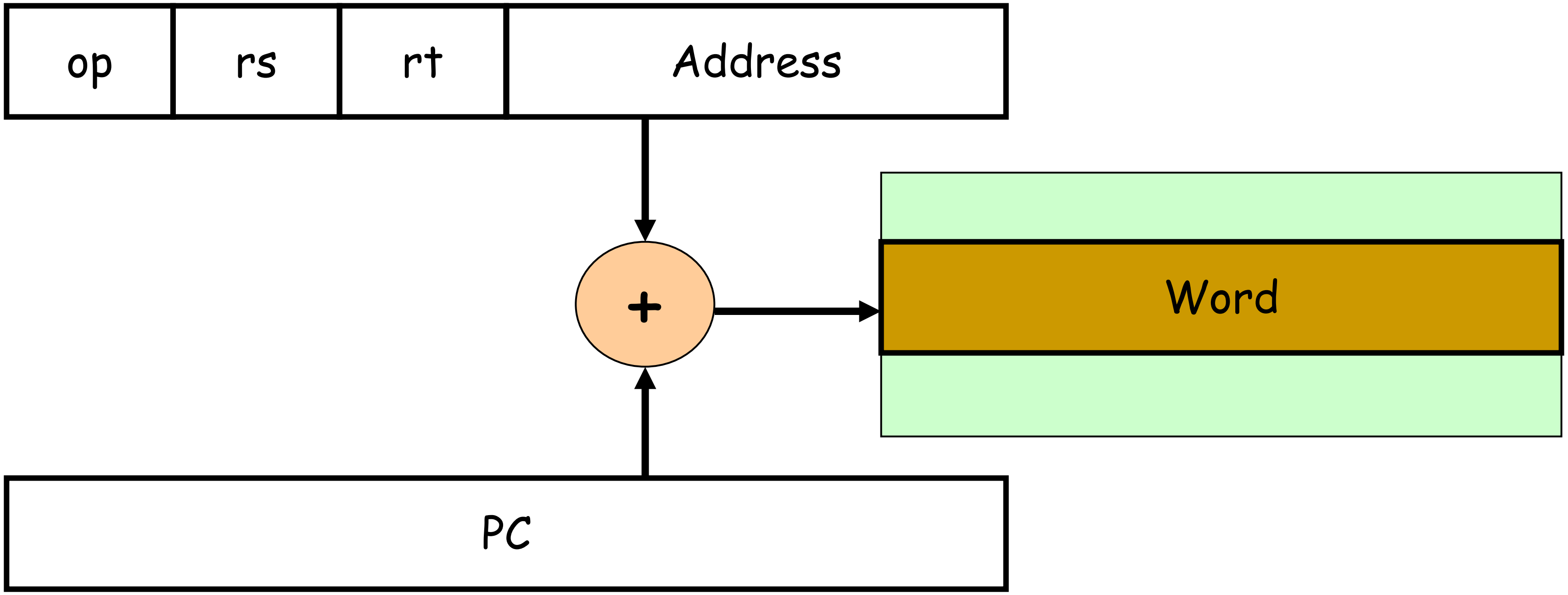
  ori $t0, $t0, 1010101010101010



| | | |
|---|---|---|
| | 1010101010101010 | 0000000000000000 |
| **ori** | 0000000000000000 | 1010101010101010 |
| | 1010101010101010 | 1010101010101010 |

# MIPS Addressing Mode (4)

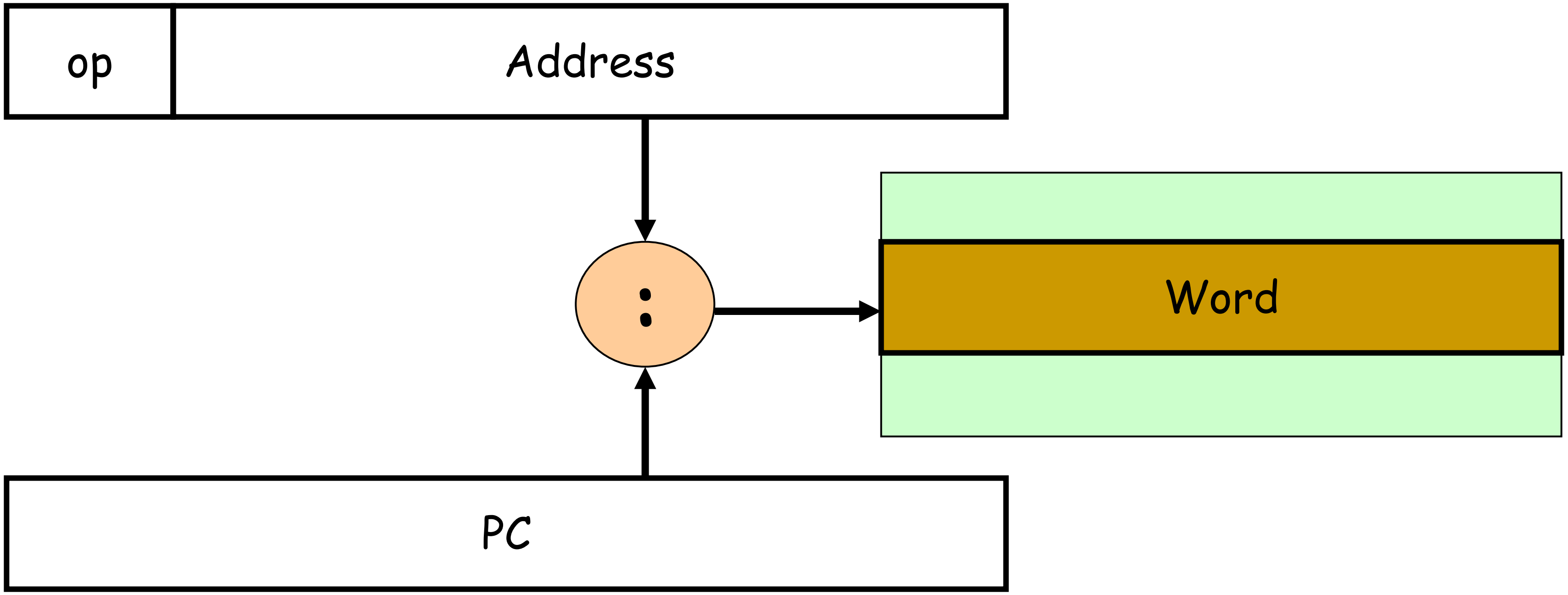
- PC-relative addressing



Example : beq $2, $3, 100

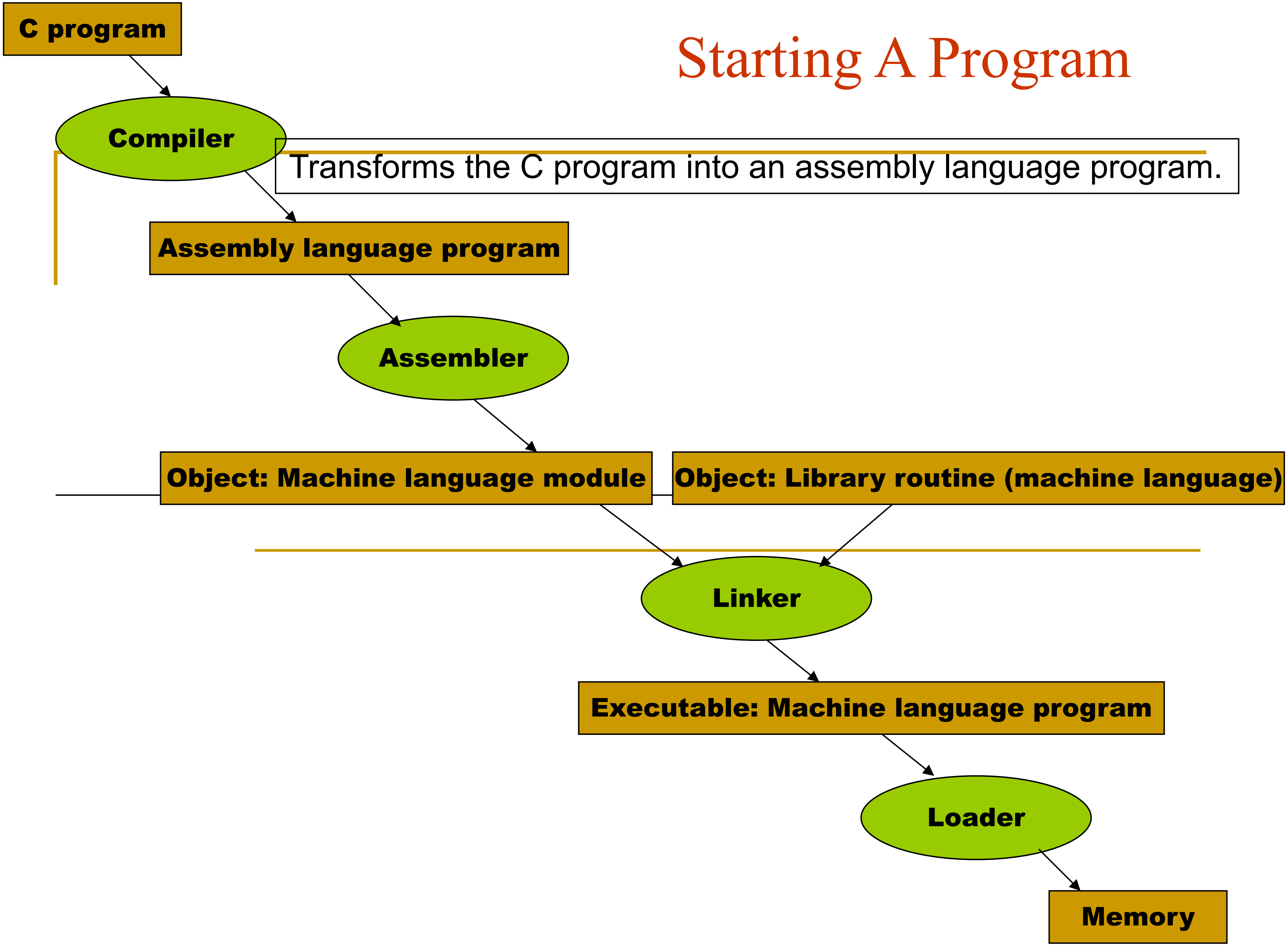
# MIPS Addressing Mode (5)

- Pseudodirect addressing



Example : j 100

# Starting A Program



C program

Compiler

Transforms the C program into an assembly language program.

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory

# Assembler

- Assembler
  - The assembler turns the assembly language program into an object file.
  - Symbol table: A table that matches names of labels to the addresses of the memory words that instruction occupy.

| **Object file header** | | | |
|---|---|---|---|
| | Name | Procedure A | |
| | Text size | 100hex | |
| | Data size | 20hex | |
| Text segment | Address | Instruction | |
| | 0 | lw $a0, 0($gp) | |
| | 4 | jal 0 | |
| | … | … | |
| Data segment | O | (X) | |
| | … | … | |
| Relocation information | Address | Instruction Type | Dependency |
| | 0 | lw | X |
| | 4 | jal | B |
| Symbol table | Label | Address | |
| | X | — | |
| | B | — | |

```
lw $a0, x
jal   B
```

# Assembler (cont.)

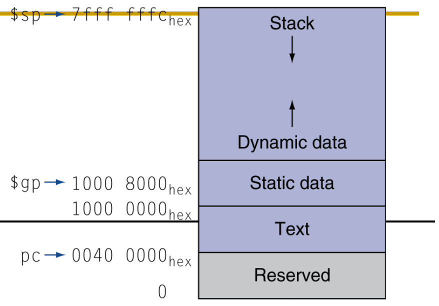
- Psudoinstruction: a common variation of assembly language instructions often treated as if it were an instruction in its own right.
  - move $t0, $t1 -> add $t0, $zero, $t1
  - blt -> slt & bne

# Linker (Link editor)

- Linker takes all the independently assembled machine language programs and “stitches” them together to produce an executable file that can be run on a computer.
- There are three steps for the linker:
  1. Place code and data modules symbolically in memory.
  2. Determine the addresses of data and instruction labels.
  3. Patch both the internal and external references.

| Object file header | | | |
|---|---|---|---|
| | Name | Procedure A | |
| | Text size | 100hex | |
| | Data size | 20hex | |
| Text segment | Address | Instruction | |
| | 0 | lw $a0, 0($gp) | |
| | 4 | jal 0 | |
| | ... | ... | |
| Data segment | 0 | (X) | |
| | ... | ... | |
| Relocation information | Address | Instruction Type | Dependency |
| | 0 | lw | X |
| | 4 | jal | B |
| Symbol table | Label | Address | |
| | X | — | |
| | B | — | |

| Object file header | | | |
|---|---|---|---|
| | Name | Procedure B | |
| | Text size | 200hex | |
| | Data size | 30hex | |
| Text segment | Address | Instruction | |
| | 0 | sw $a1, 0($gp) | |
| | 4 | jal 0 | |
| | ... | ... | |
| Data segment | 0 | (Y) | |
| | ... | ... | |
| Relocation information | Address | Instruction Type | Dependency |
| | 0 | sw | Y |
| | 4 | jal | A |
| Symbol table | Label | Address | |
| | Y | — | |
| | A | — | |



| Executable file header | | |
|---|---|---|
| | Text size | 300hex |
| | Data size | 50hex |
| Text segment | Address | Instruction |
| A | 0040 0000hex | lw $a0, 8000hex ($gp) |
| | 0040 0004hex | jal 40 0100hex |
| | … | … |
| B | 0040 0100hex | sw $a1, 8020hex ($gp) |
| | 0040 0104hex | jal 40 0000hex |
| | … | … |
| Data segment | Address | |
| | 1000 0000hex | (X) |
| | … | … |
| | 1000 0020hex | (Y) |
| | … | … |

sw $a0, y
jal A

# Loader

- Read the executables file header to determine the size of the text and data segments
- Creates an address space large enough for the text and data
- Copies the instructions and data from the executable file into memory
- Copies the parameters (if any) to the main program onto the stack
- Initializes the machine registers and sets the stack pointer the first free location
- Jump to a start-up routine

```
 main();

_start_up:
   lw   a0, offset($sp)   ## load arguments
   jal   main;
   exit
```

# Dynamically Linked Libraries (DLL)

- Disadvantages with traditional statically linked library
  - Library updates
  - Loading the whole library even if all of the library is not used
- Dynamically linked library
  - The libraries are not linked and loaded until the program is run.
  - Lazy procedure linkage
    - Each routine is linked only after it is called.

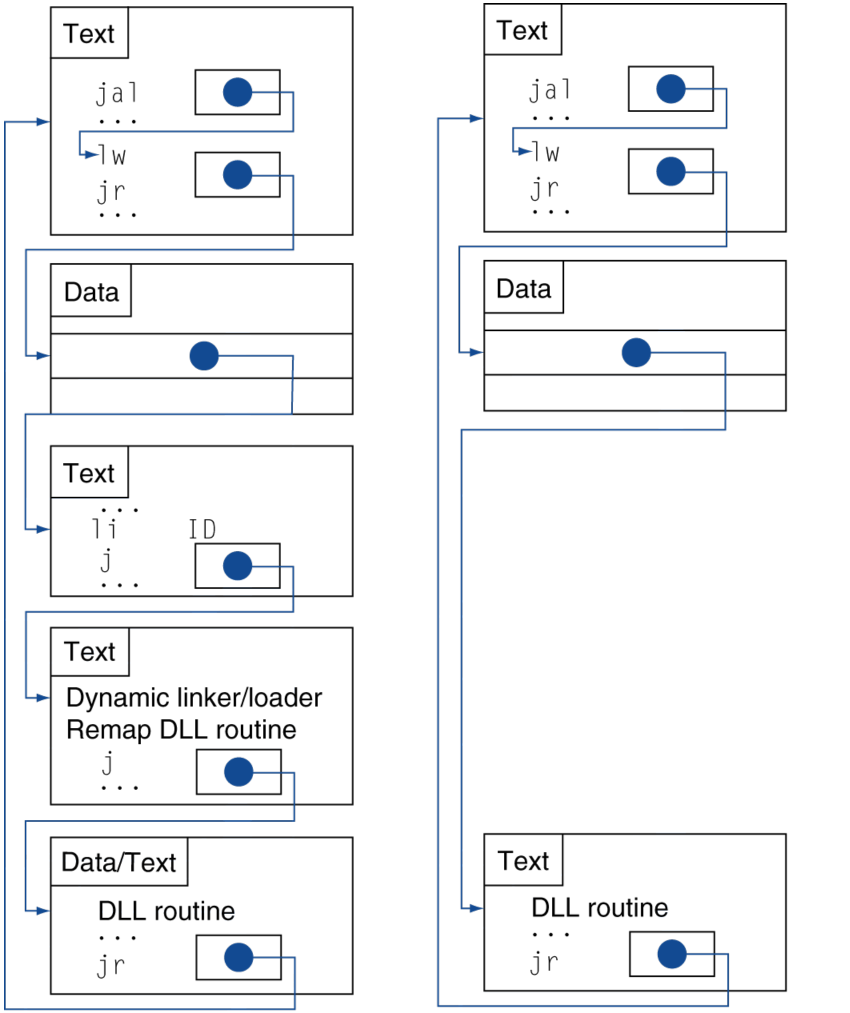
# Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

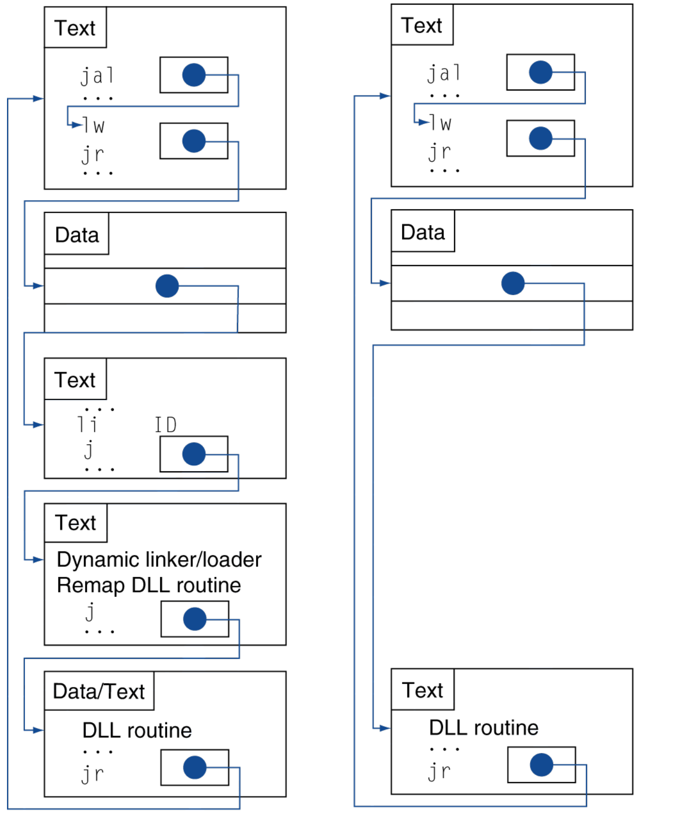
Linker/loader code

Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine



a. First call to DLL routine

b. Subsequent calls to DLL routine

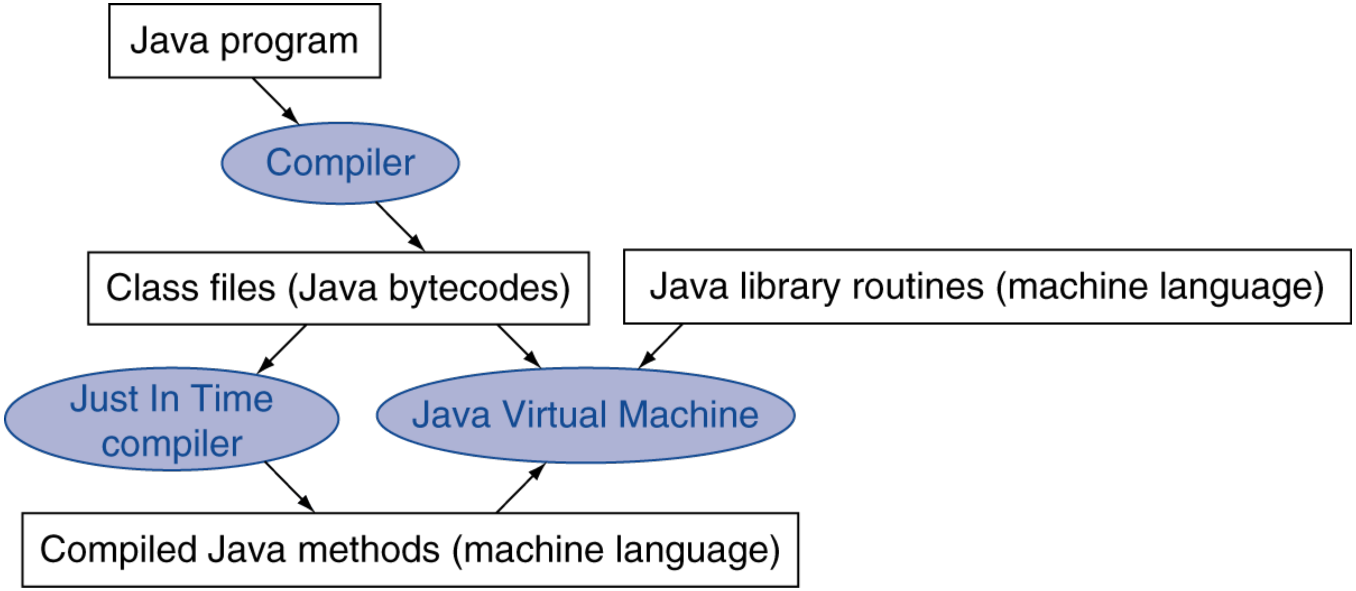
Text
jal printf();
printf ()
{ lw  $r1,  printf_addr
  jr    $r1}

Data
printf_addr .word L1 → Data
printf_addr 0x400000

Text
L1:  li ID
       j DLL;

Text
printf()
:::::::
jr $ra
→ Assume this is loaded into 0x400000

# Starting a Java Program


Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)

**Java bytecode:** Instruction from an instruction set designed to interpret Java programs.

**Java Virtual Machine (JVM):** The program that interprets Java bytecodes

**Just In Time Compiler (JIT):** A compiler that operates at runtime, translating the interpreted code segments into the native code of the compiler.

# ARM & MIPS Similarities

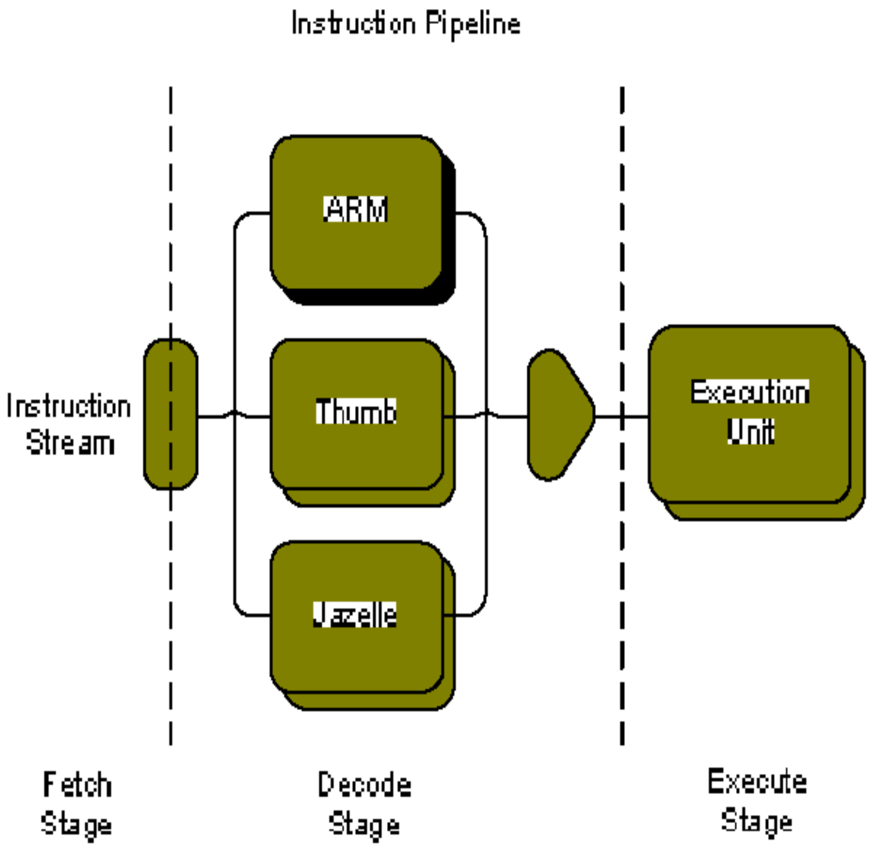
§2.16 Real Stuff: ARM Instructions

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

| | ARM | MIPS |
|---|---|---|
| Date announced | 1985 | 1985 |
| Instruction size | 32 bits | 32 bits |
| Address space | 32-bit flat | 32-bit flat |
| Data alignment | Aligned | Aligned |
| Data addressing modes | 9 | 3 |
| Registers | 15 × 32-bit | 31 × 32-bit |
| Input/output | Memory mapped | Memory mapped |

# Instruction Set

- 32-bit ARM instruction
- 16-bit Thumb instruction
- 8-bit Java Instruction Set

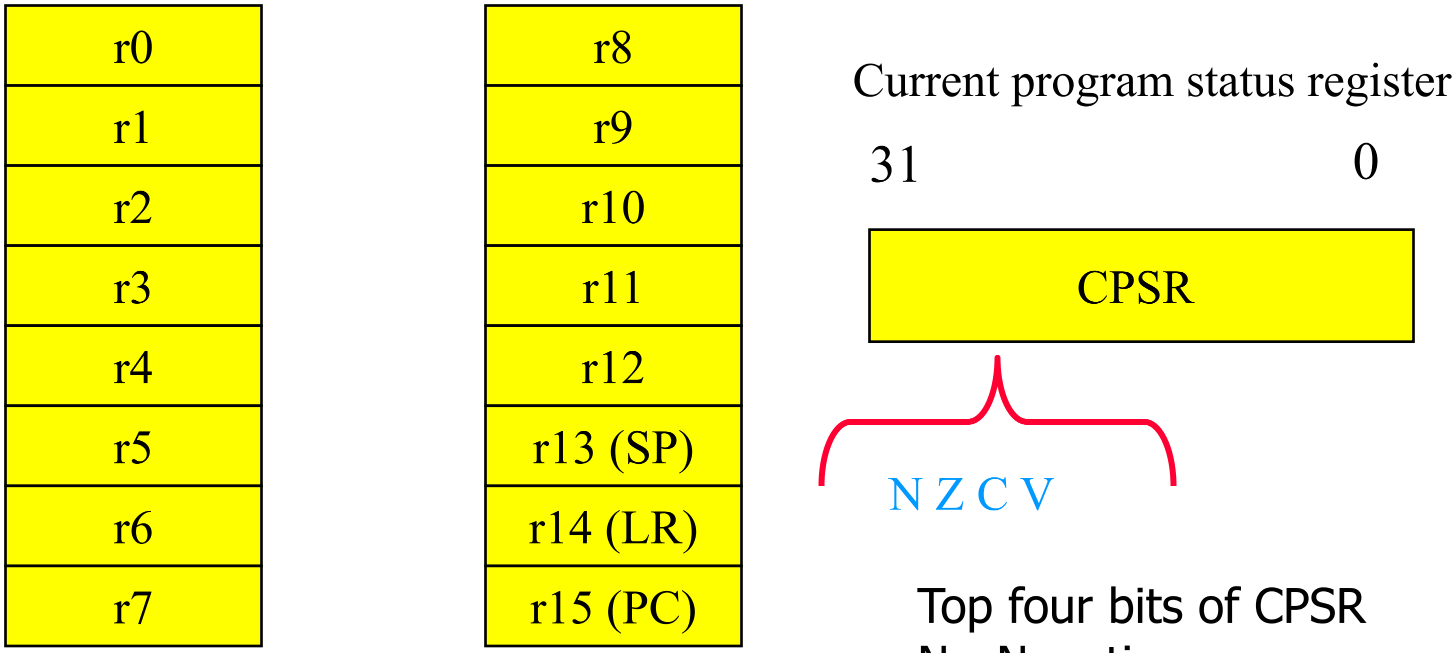


Sources: white paper on Jazelle Technology

# Main features of 32-bit Arm instruction

- Load-store architecture
- Fixed-length = 32 bits
- 3-address instruction format (2 sources, 1 result operands)
- DSP instruction
  - Support MAC & SIMD operations
- Conditional execution of ALL instructions

# Registers



r0
r1
r2
r3
r4
r5
r6
r7

r8
r9
r10
r11
r12
r13 (SP)
r14 (LR)
r15 (PC)

Current program status register

31 0

CPSR

N Z C V

Top four bits of CPSR
N : Negative
Z : Zero
C : Carry
V : Overflow

# Conditional Execution

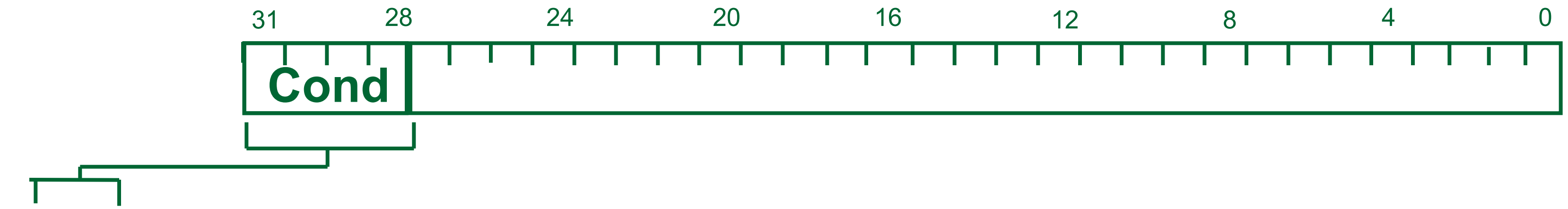
- All instructions contain a condition field which determines whether the CPU will execute them.
  - Allows very dense in-line code, without branches.
  - The time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

  - For example an add instruction takes the form:
    - `ADD r0,r1,r2      ; r0 = r1 + r2`

  - To execute this only if the zero flag is set:

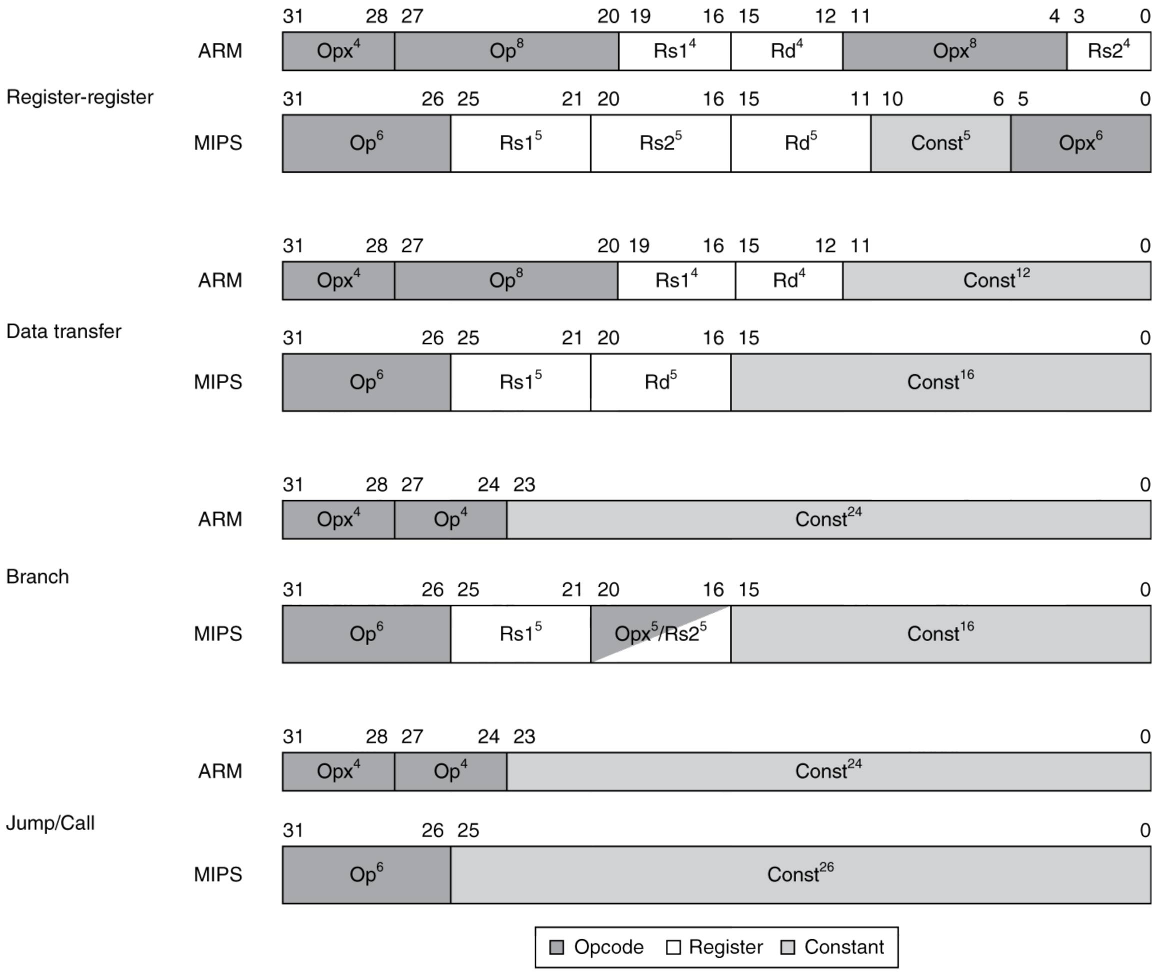
    -
      ```
      ADDEQ r0,r1,r2    ; If zero flag set then…
                        ; ... r0 = r1 + r2
      ```

# The Condition Field



| Code | Suffix | Description | Flags |
|---|---|---|---|
| 0000 | EQ | Equal | Z=1 |
| 0001 | NE | Not equal | Z=0 |
| 0010 | CS/HS | Unsigned higher or same | C=1 |
| 0011 | CC/LO | Unsigned lower | C=0 |
| 0100 | MI | Minus | N=1 |
| 0101 | PL | Positive or Zero | N=0 |
| 0110 | VS | Overflow | V=1 |
| 0111 | VC | No overflow | V=0 |
| 1000 | HI | Unsigned higher | C=1 & Z=0 |
| 1001 | LS | Unsigned lower or same | C=0 or Z=1 |
| 1010 | GE | Greater or equal | N=V |
| 1011 | LT | Less than | N!=V |
| 1100 | GT | Greater than | Z=0 & N=V |
| 1101 | LE | Less than or equal | Z=1 or N=!V |
| 1110 | AL | Always | none |

# Instruction Encoding



Register-register

ARM
31 28 27 20 19 16 15 12 11 4 3 0
Opx$^4$ | Op$^8$ | Rs1$^4$ | Rd$^4$ | Opx$^8$ | Rs2$^4$

MIPS
31 26 25 21 20 16 15 11 10 6 5 0
Op$^6$ | Rs1$^5$ | Rs2$^5$ | Rd$^5$ | Const$^5$ | Opx$^6$

Data transfer

ARM
31 28 27 20 19 16 15 12 11 0
Opx$^4$ | Op$^8$ | Rs1$^4$ | Rd$^4$ | Const$^{12}$

MIPS
31 26 25 21 20 16 15 0
Op$^6$ | Rs1$^5$ | Rd$^5$ | Const$^{16}$

Branch

ARM
31 28 27 24 23 0
Opx$^4$ | Op$^4$ | Const$^{24}$

MIPS
31 26 25 21 20 16 15 0
Op$^6$ | Rs1$^5$ | Opx$^5$/Rs2$^5$ | Const$^{16}$

Jump/Call

ARM
31 28 27 24 23 0
Opx$^4$ | Op$^4$ | Const$^{24}$

MIPS
31 26 25 0
Op$^6$ | Const$^{26}$

■ Opcode □ Register ▨ Constant

# The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

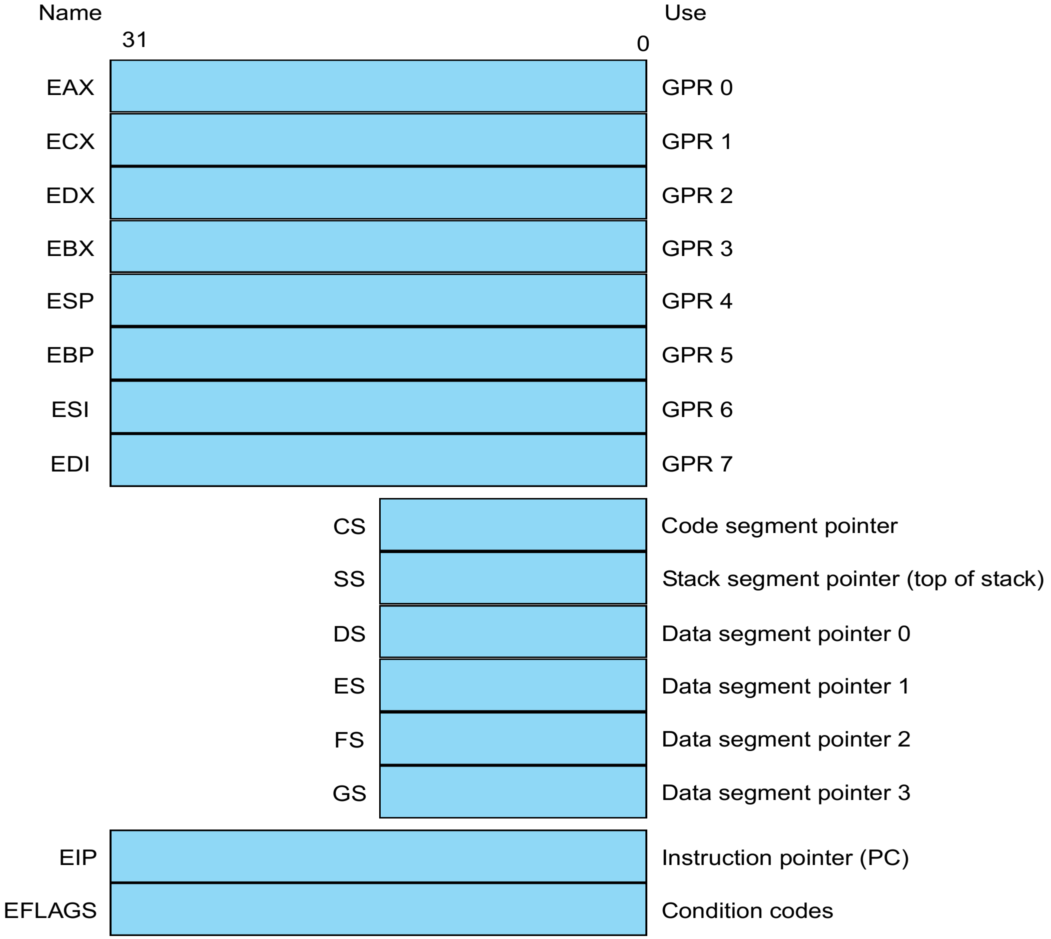
# The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers (128 bits)
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

# The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers (128bit->256 bits) , more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

# IA-32 Registers

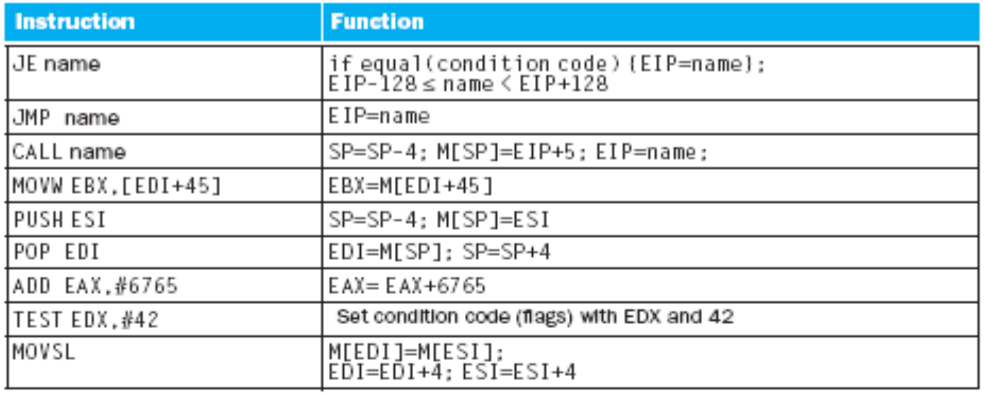


| Name | Use |
|---|---|
| EAX | GPR 0 |
| ECX | GPR 1 |
| EDX | GPR 2 |
| EBX | GPR 3 |
| ESP | GPR 4 |
| EBP | GPR 5 |
| ESI | GPR 6 |
| EDI | GPR 7 |
| CS | Code segment pointer |
| SS | Stack segment pointer (top of stack) |
| DS | Data segment pointer 0 |
| ES | Data segment pointer 1 |
| FS | Data segment pointer 2 |
| GS | Data segment pointer 3 |
| EIP | Instruction pointer (PC) |
| EFLAGS | Condition codes |

IA-32 has only 8 general purpose register vs. 32 in MIPS

# 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

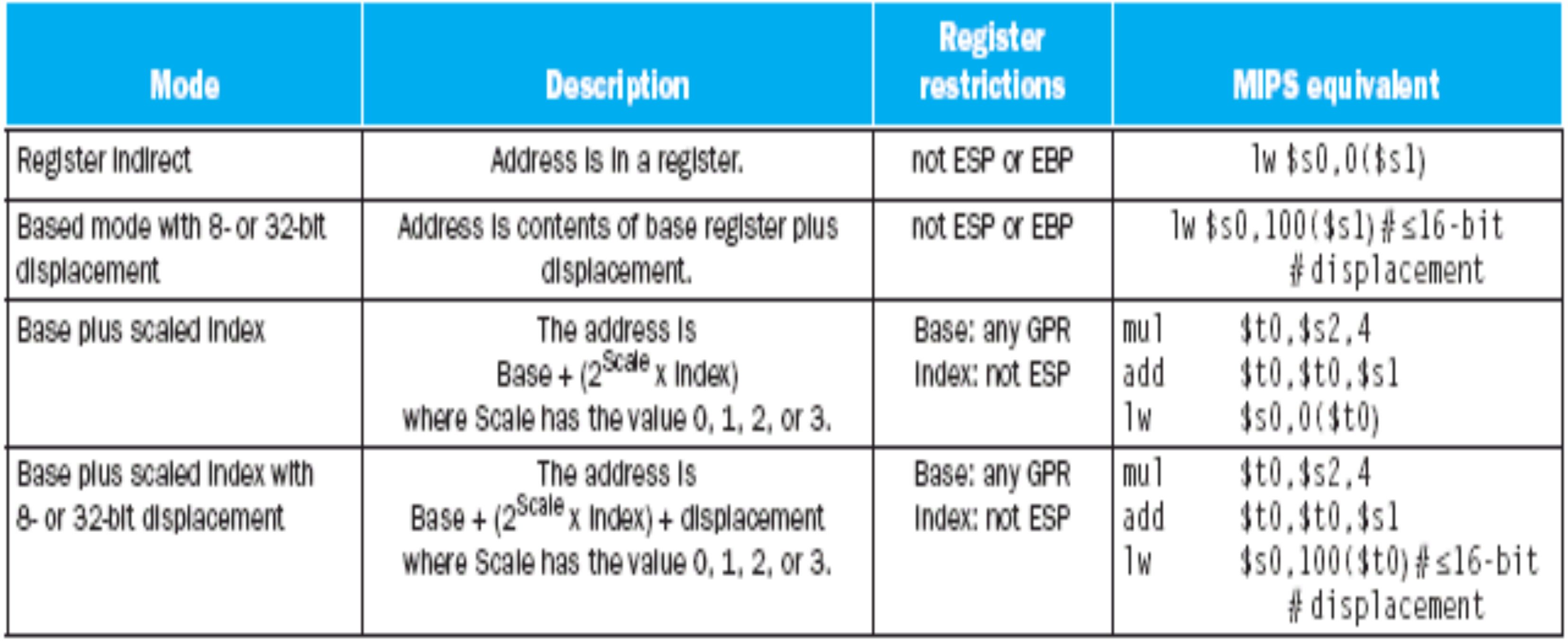


| Instruction | Function |
|---|---|
| JE name | if equal(condition code) {EIP=name};<br>EIP-128 ≤ name < EIP+128 |
| JMP  name | EIP=name |
| CALL name | SP=SP-4; M[SP]=EIP+5; EIP=name; |
| MOVW EBX,[EDI+45] | EBX=M[EDI+45] |
| PUSH ESI | SP=SP-4; M[SP]=ESI |
| POP  EDI | EDI=M[SP]; SP=SP+4 |
| ADD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

1. IA-32: Two or three-operand operation vs. MIPS: three-operand operation
2. IA-32: Register-memory vs. MIPS: register-register

# IA-32 Addressing Mode

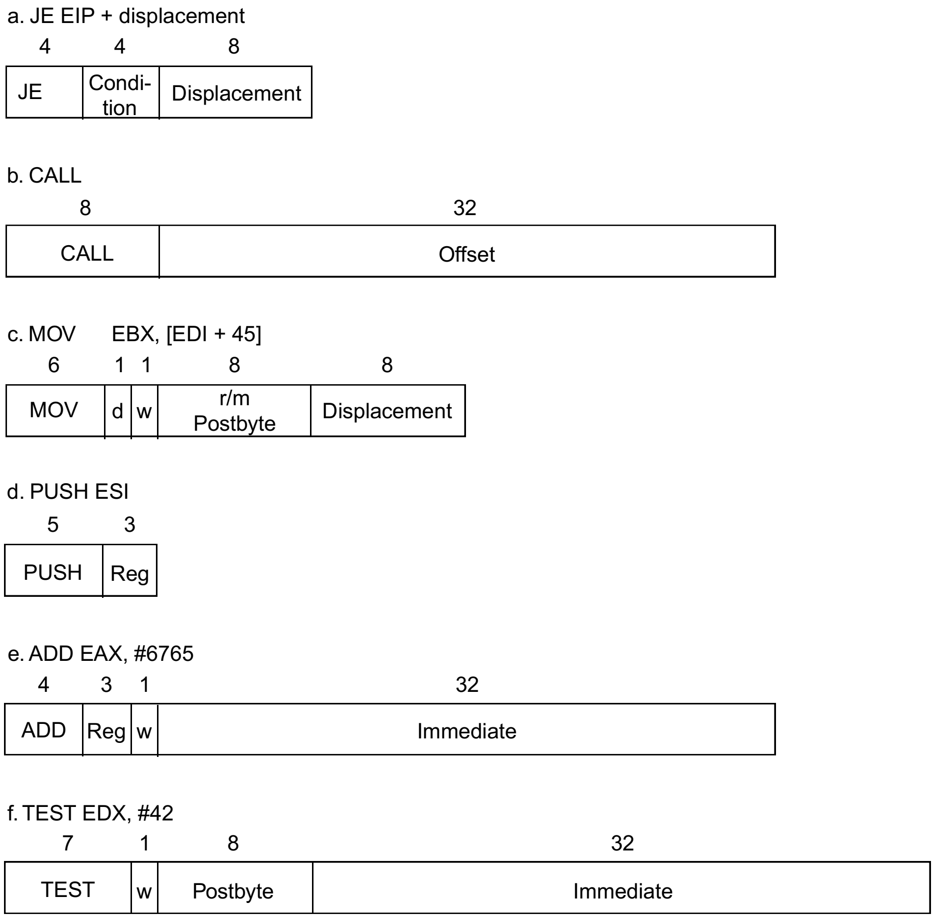


| Mode | Description | Register restrictions | MIPS equivalent |
|---|---|---|---|
| Register indirect | Address is in a register. | not ESP or EBP | `lw $s0,0($s1)` |
| Based mode with 8- or 32-bit displacement | Address is contents of base register plus displacement. | not ESP or EBP | `lw $s0,100($s1)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

Move EAX, [EBX+ EPI*4 + 100]

- IA-32 has more addressing modes than MIPS

# IA-32 instruction Formats



a. JE EIP + displacement

| 4 | 4 | 8 |
|---|---|---|
| JE | Condi-tion | Displacement |

b. CALL

| 8 | 32 |
|---|---|
| CALL | Offset |

c. MOV EBX, [EDI + 45]

| 6 | 1 | 1 | 8 | 8 |
|---|---|---|---|---|
| MOV | d | w | r/m<br>Postbyte | Displacement |

d. PUSH ESI

| 5 | 3 |
|---|---|
| PUSH | Reg |

e. ADD EAX, #6765

| 4 | 3 | 1 | 32 |
|---|---|---|---|
| ADD | Reg | w | Immediate |

f. TEST EDX, #42

| 7 | 1 | 8 | 32 |
|---|---|---|---|
| TEST | w | Postbyte | Immediate |

IA-32 variable-length encoding vs. MIPS fixed-length encoding

# Implementing IA-32

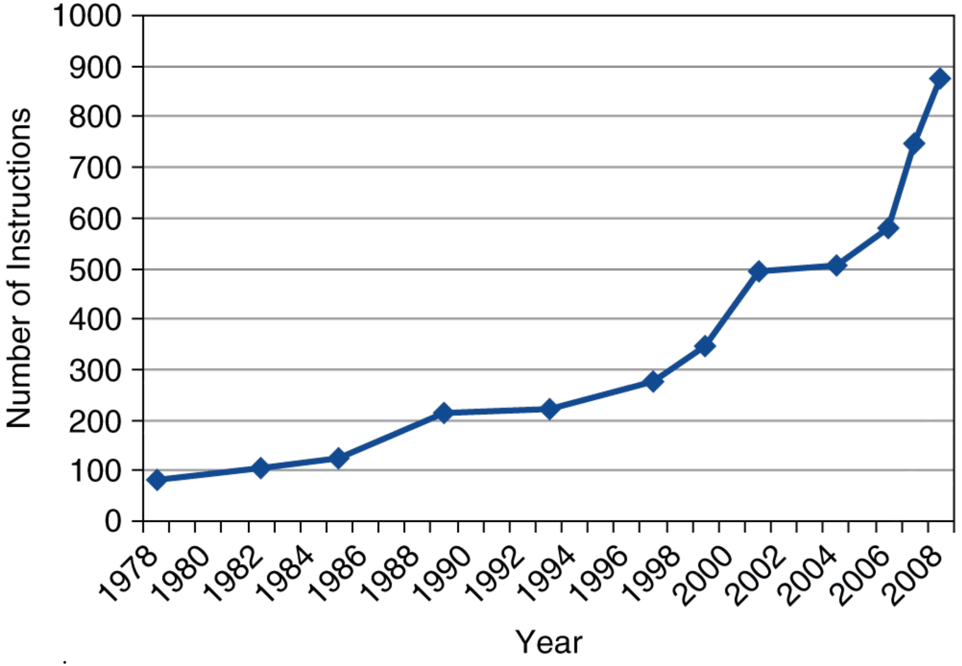
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

# Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

# Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

# Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to a local variable after procedure returns
  - Pointer becomes invalid when stack popped

# Concluding Remarks

§2.19 Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

# Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

| Instruction class | MIPS examples | SPEC2006 Int | SPEC2006 FP |
|---|---|---|---|
| Arithmetic | add, sub, addi | 16% | 48% |
| Data transfer | lw, sw, lb, lbu, lh, lhu, sb, lui | 35% | 36% |
| Logical | and, or, nor, andi, ori, sll, srl | 12% | 4% |
| Cond. Branch | beq, bne, slt, slti, sltiu | 34% | 8% |
| Jump | j, jr, jal | 2% | 0% |