Instructions: Language of the Computer (2)

Jo, Heeseung

Procedure Calling

Steps required

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call

Register Usage

$a0 - $a3: arguments (reg's 4 - 7)

$v0, $v1: result values (reg's 2 and 3)

$t0 - $t9: temporaries

- Can be overwritten by callee

$s0 - $s7: saved

- Must be saved/restored by callee

$gp: global pointer for static data (reg 28)

$sp: stack pointer (reg 29)

$fp: frame pointer (reg 30)

$ra: return address (reg 31)

Procedure Call Instructions

Procedure call: jump and link

```
jal ProcedureLabel
```

- Put the address of following instruction into $ra
- Jumps to target address

Procedure return: jump register

```
jr $ra
```

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

Leaf Procedure Example

C code:

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

- Arguments g, …, j in $a0, …, $a3
- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0

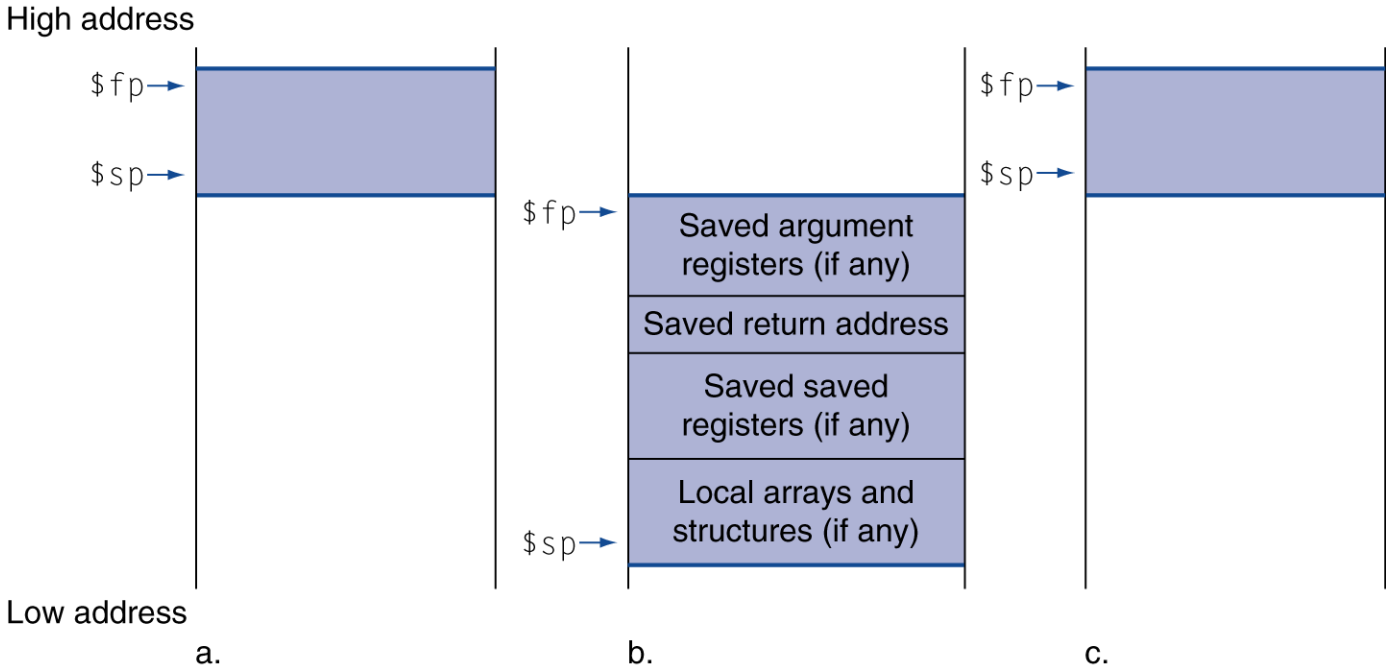
Non-Leaf Procedures

Procedures that call other procedures

For nested call, caller needs to save on the stack:

- Its return address
- Any arguments and temporaries needed after the call

Restore from the stack after the call

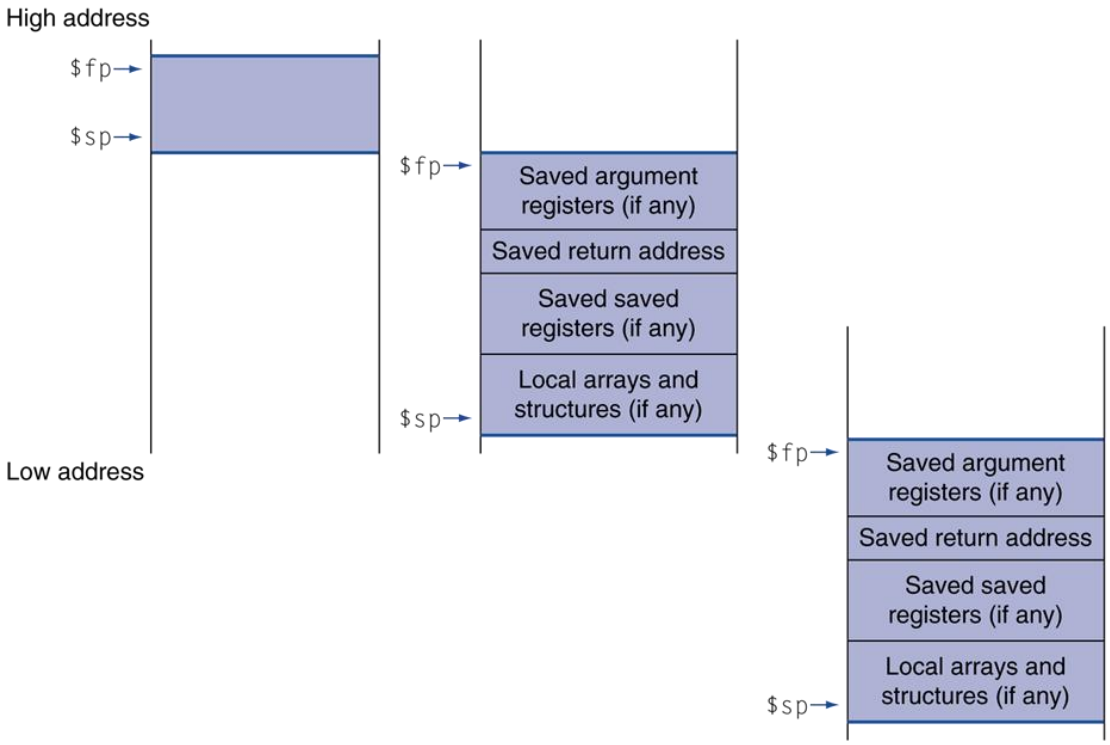


Non-Leaf Procedure Example

C code:

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

- Argument n in $a0
- Result in $v0



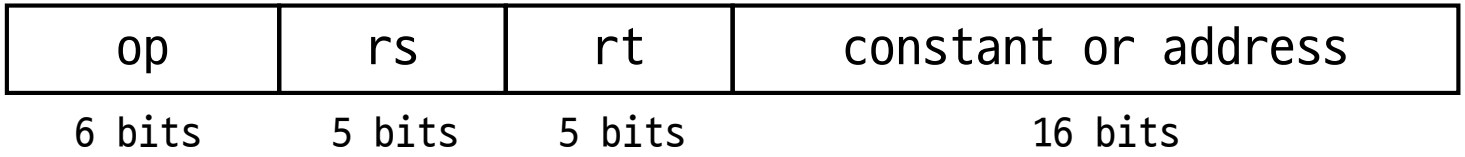
Branch Addressing

Branch instructions specify

- Opcode, two registers, target address
- beq rs, rt, L1

Most branch targets are near branch

- Forward or backward



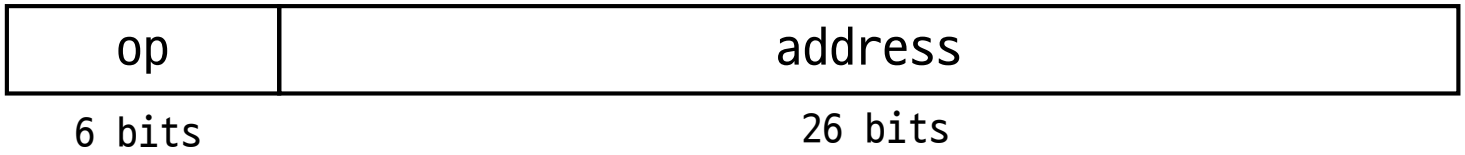
PC-relative addressing

- Target address = PC + offset × 4
- PC already incremented by 4 by this time

Jump Addressing

Jump (j and jal) targets could be anywhere in text segment

- Encode full address in instruction



(Pseudo)Direct jump addressing

- Target address = $PC_{31\ldots28}$: (address × 4)

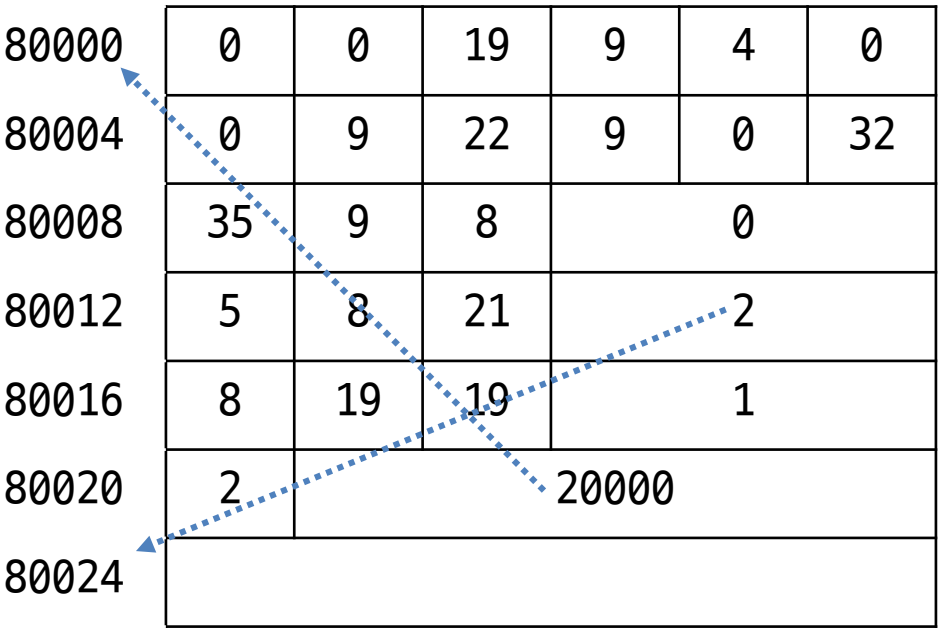
Target Addressing Example

Loop code from earlier example

- Assume Loop at location 80000

$t0 - $t7 are reg's 8 - 15
$t8 - $t9 are reg's 24 - 25
$s0 - $s7 are reg's 16 - 23

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



Addressing Mode Summary

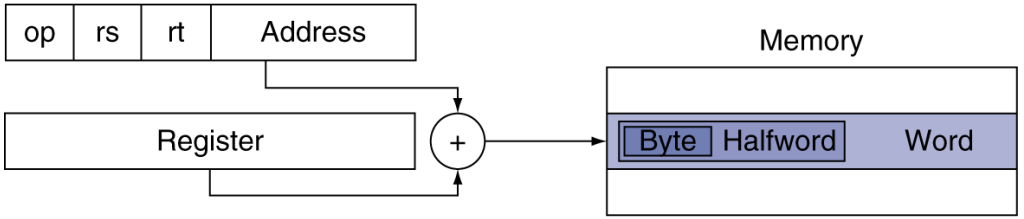
1. Immediate addressing


```
addi $s3, $s3, 4
```

2. Register addressing

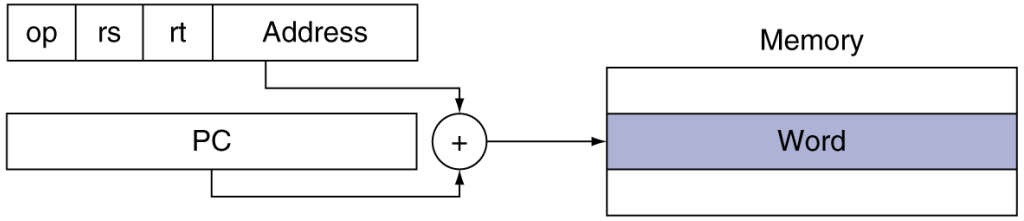

```
add $t0, $s3, $s3
```

3. Base addressing



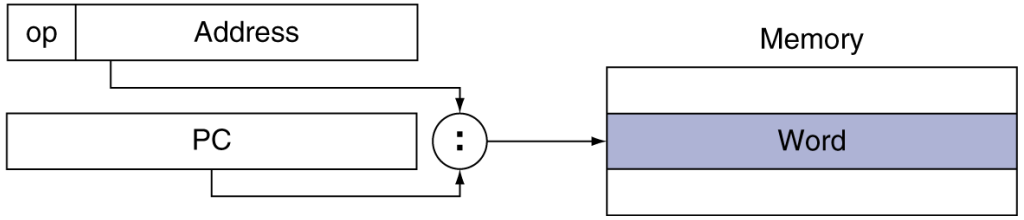

```
lw  $t0, 32($s3)
```

4. PC-relative addressing




```
bne  $t0, $s5, Exit
```

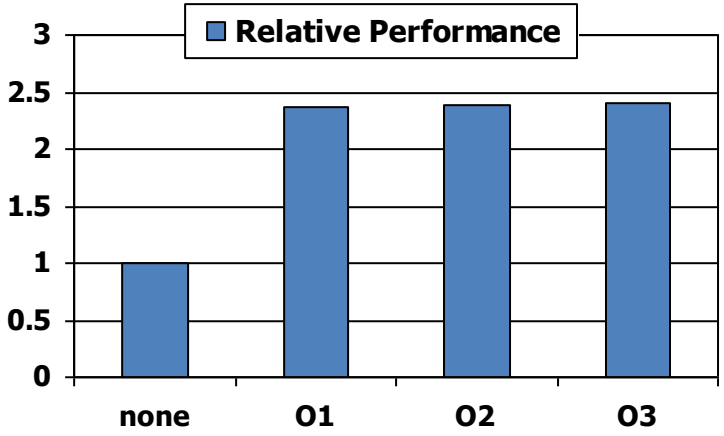
5. Pseudodirect addressing

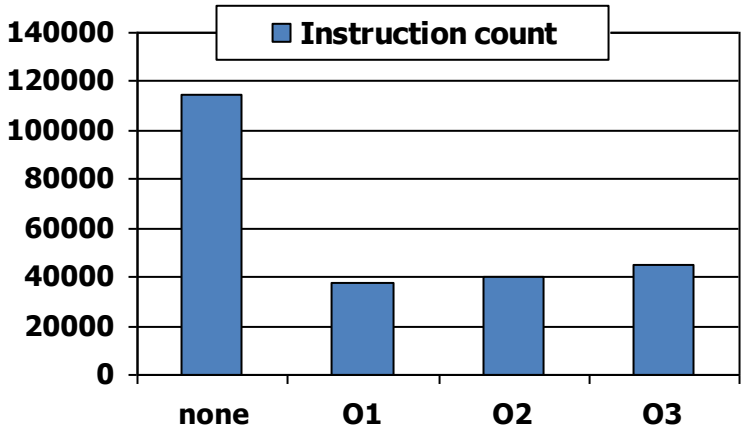


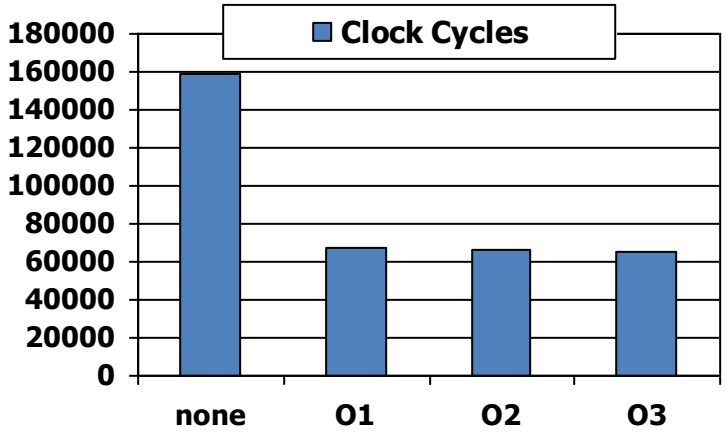

```
j    Loop
```

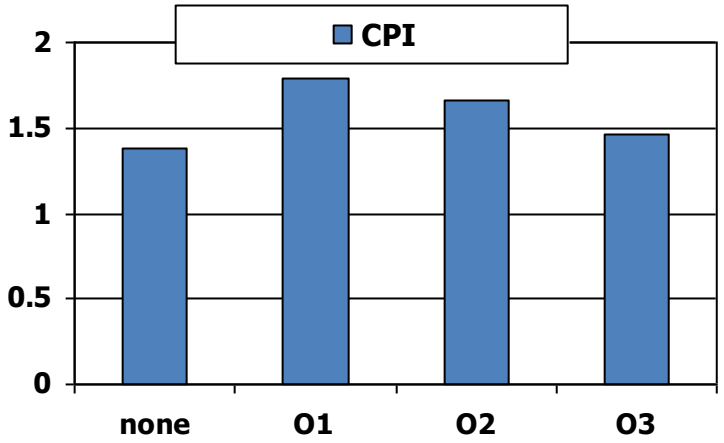
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

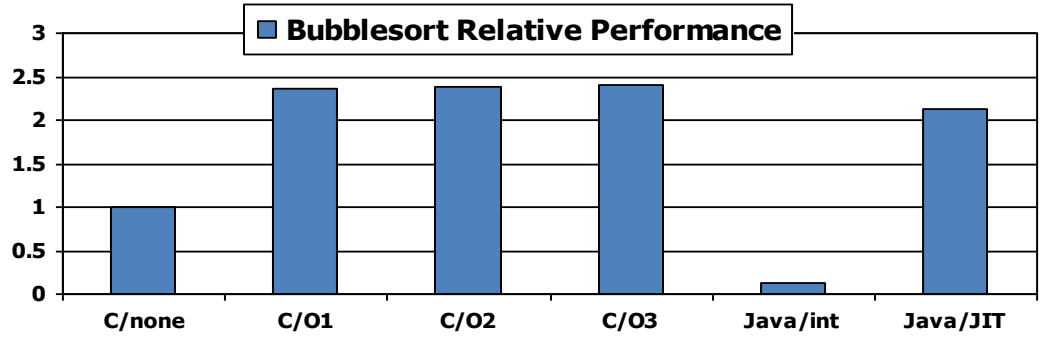


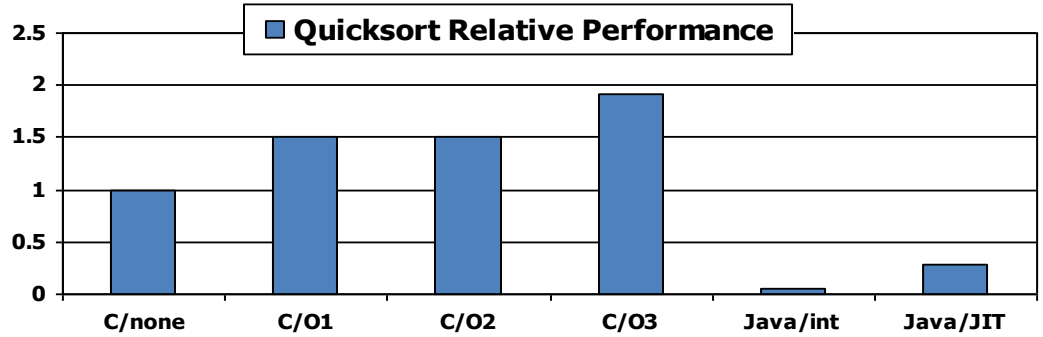


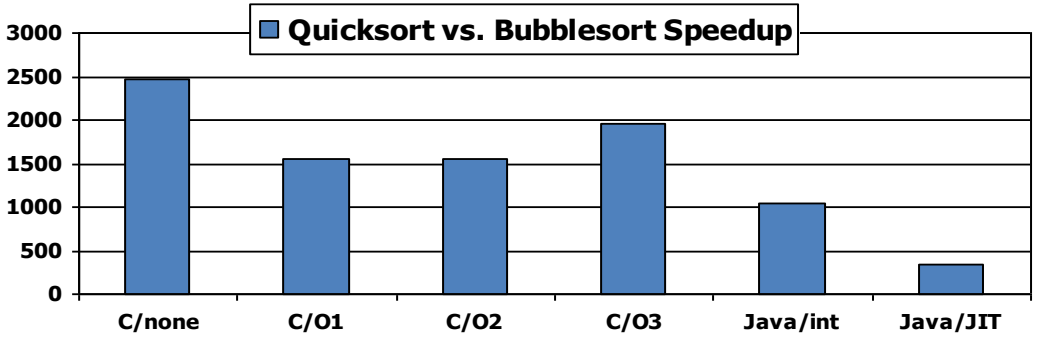




Effect of Language and Algorithm







Lessons Learnt

Instruction count and CPI are not good performance indicators in isolation

Compiler optimizations are sensitive to the algorithm

Java/JIT compiled code is significantly faster than JVM interpreted

- Comparable to optimized C in some cases

Nothing can fix a dumb algorithm!

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
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions

The Intel x86 ISA

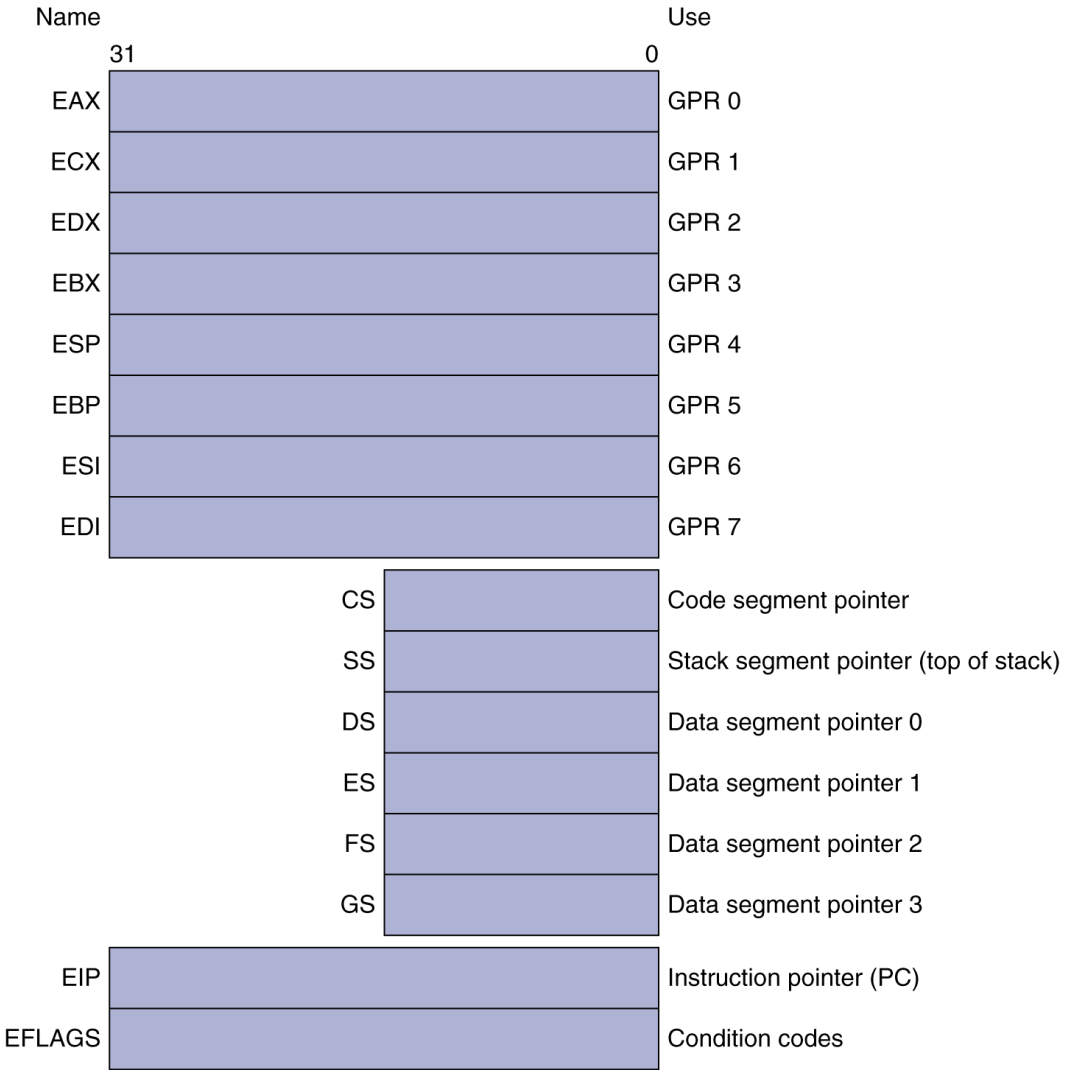
And further ...

- 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
- Advanced Vector Extension (announced 2008)
  - Longer SSE registers, more instructions

If Intel didn't extend with compatibility, its competitors would!

- Technical elegance ≠ market success

Basic x86 Registers



Basic x86 Addressing Modes

Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

c.f) MIPS has no memory operand (use load, store)

Basic x86 Addressing Modes

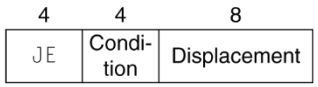
movl operand combinations

- Cannot do memory-memory transfers with single instruction

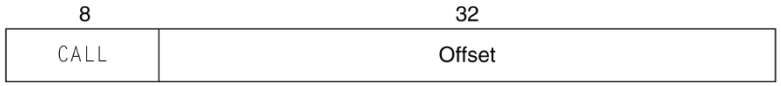
	Source	Destination		C Analog
movl	*Imm*	*Reg*	movl $0x4,%eax	temp = 0x4;
		Mem	movl $-147,(%eax)	*p = -147;
	Reg	*Reg*	movl %eax,%edx	temp2 = temp1;
		Mem	movl %eax,(%edx)	*p = temp;
	Mem	*Reg*	movl (%eax),%edx	temp = *p;

x86 Instruction Encoding

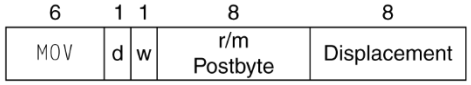
a. JE EIP + displacement



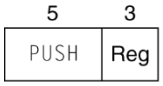
b. CALL



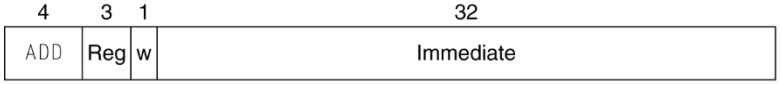
c. MOV EBX, [EDI + 45]



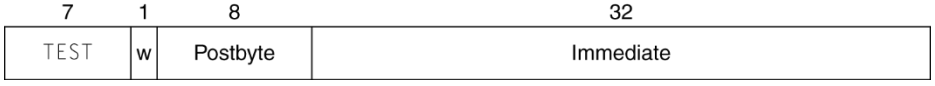
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
  - Operand length, repetition, locking, …

Fallacies

Powerful instruction ⇒ higher performance

- It is not always true
- 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

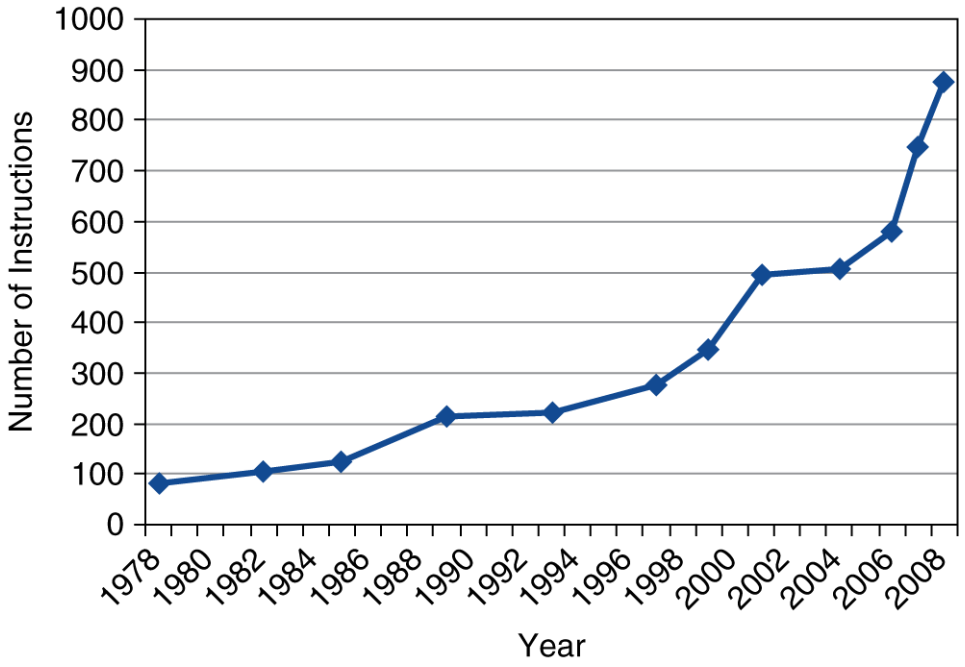
Use assembly code for high performance

- 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

Concluding Remarks

Design principles

- Simplicity favors regularity
- Smaller is faster
- Make the common case fast
- 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%