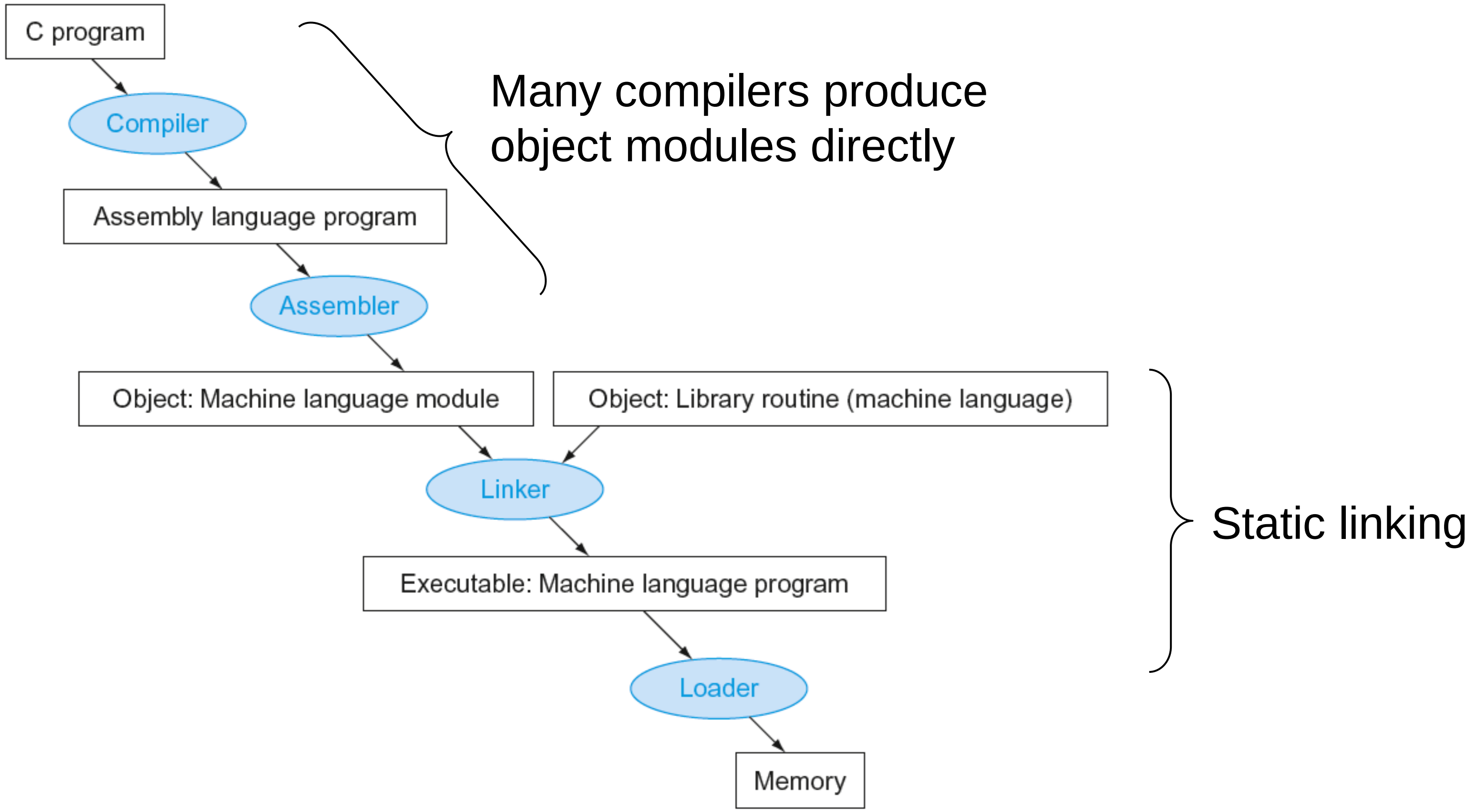
TURIN POLYTECHNIC
UNIVERSITY IN
TASHKENT

Fundamentals of Computer Architecture

ARM - Review

Teacher: Lobar Asretdinova

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Many compilers produce object modules directly

Static linking

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including SP, FP)
  6. Jump to startup routine
     - Copies arguments to X0, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

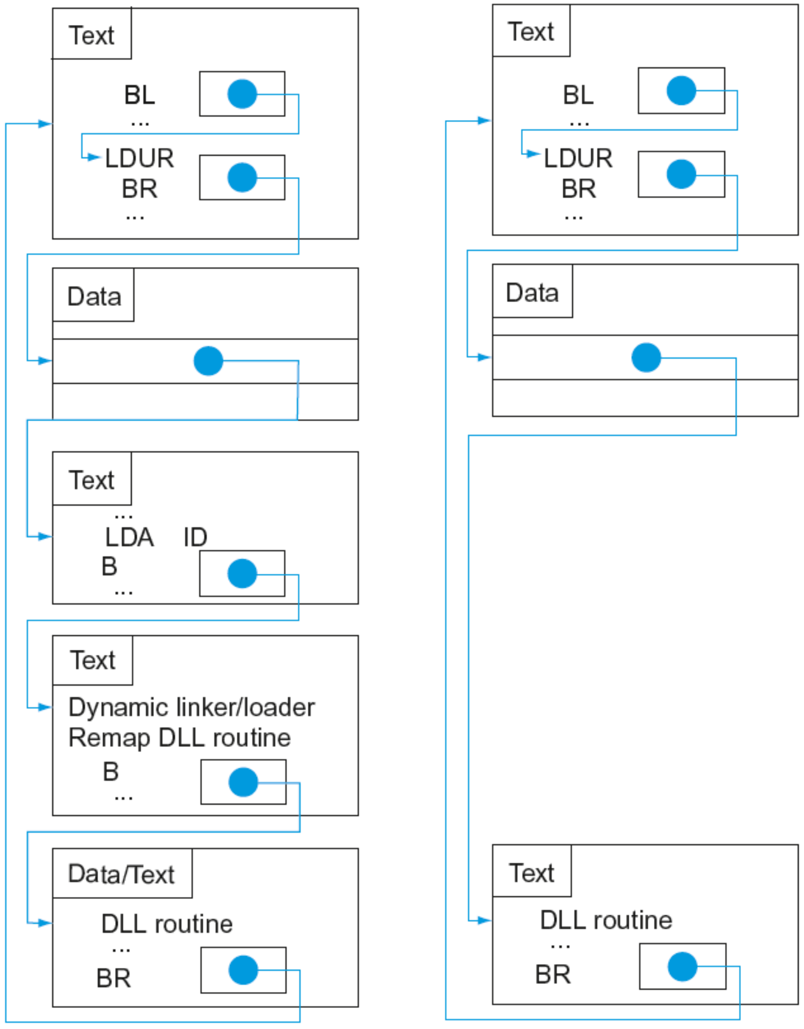
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

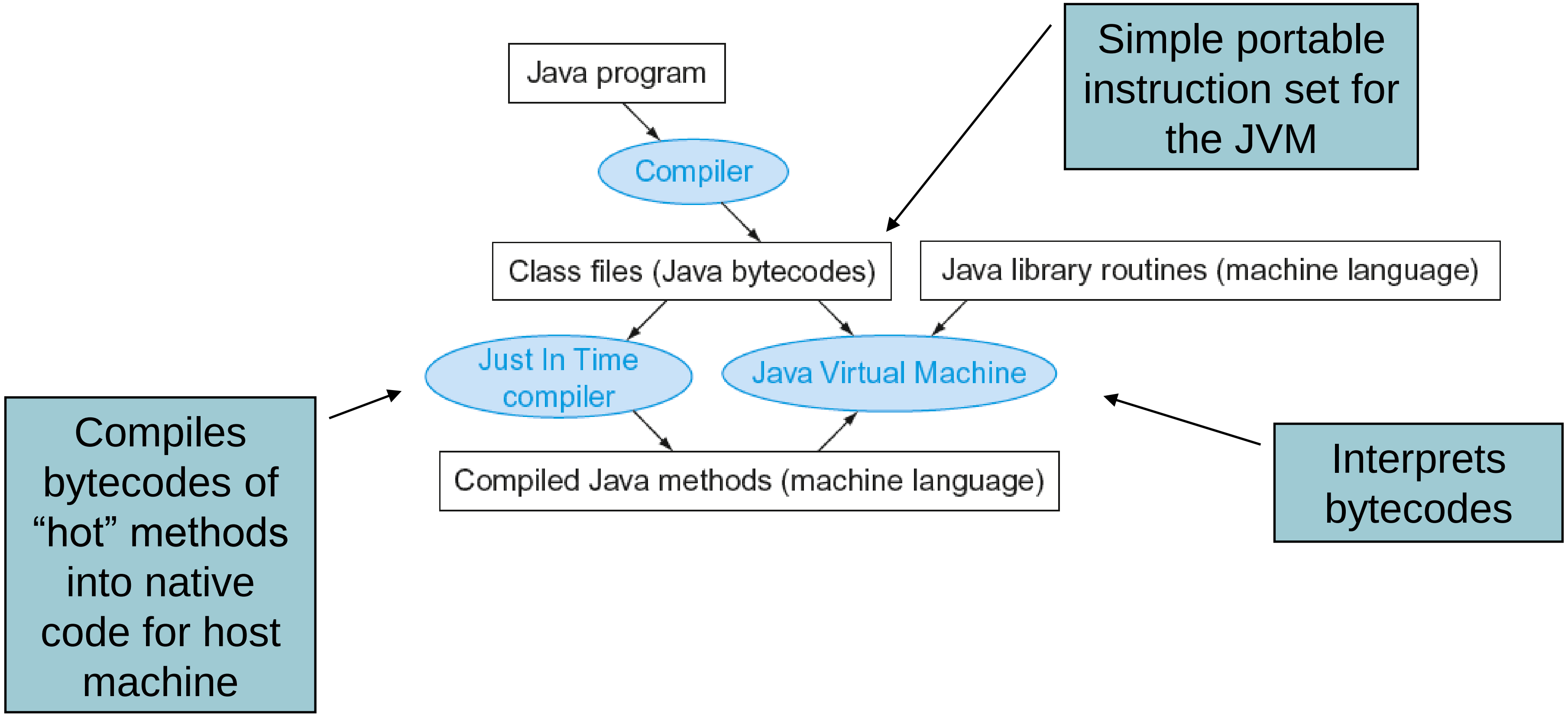
Dynamically
mapped code



(a) First call to DLL routine

(b) Subsequent calls to DLL routine

Starting Java Applications


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

Simple portable instruction set for the JVM

Compiles bytecodes of "hot" methods into native code for host machine

Interprets bytecodes

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

  - v in X0, k in X1, temp in X9

The Procedure Swap

```
swap: LSL X10,X1,#3        // X10 = k * 8
      ADD X10,X0,X10       // X10 = address of v[k]
      LDR X9,[X10,#0]      // X9 = v[k]
      LDR X11,[X10,#8]     // X11 = v[k+1]
      STR X11,[X10,#0]  // v[k] = X11 (v[k+1])
      STR X9,[X10,#8]   // v[k+1] = X9 (v[k])
      BR LR                // return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

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

  - v in X0, n in X1, i in X19, j in X20

n = 5

2, 7, 6, 3, 4

0 1, 2

1)

2) i = 1

j = 0

3) i = 2

j = 1

v[1] = 5

v[2] = 6

4) i = 3 j = 2

v[2] = 6 v[3] = 3

The Outer Loop

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
   MOV X19,XZR                // i = 0
for1tst:
   CMP X19, X1                // compare X19 to X1 (i to n)
   B.GE exit1                 // go to exit1 if X19 ≥ X1 (i≥n)

   (body of outer for-loop)

   ADD X19,X19,#1             // i += 1
   B for1tst                  // branch to test of outer loop
exit1:
```

The Inner Loop

- Skeleton of inner loop:
  - for (j = i − 1; j >= 0 && v[j] > v[j + 1]; j − = 1) {

```
          SUB X20, X19, #1       // j = i - 1
for2tst: CMP X20,XZR             // compare X20 to 0 (j to 0)
          B.LT exit2             // go to exit2 if X20 < 0 (j < 0)
          LSL X10, X20, #3       // reg X10 = j * 8
          ADD X11, X0, X10       // reg X11 = v + (j * 8)
          LDR X12, [X11,#0]      // reg X12 = v[j]
          LDR X13, [X11,#8]      // reg X13 = v[j + 1]
          CMP X12, X13           // compare X12 to X13
          B.LE exit2             // go to exit2 if X12 ≤ X13
          MOV X0, X21            // first swap parameter is v
          MOV X1, X20            // second swap parameter is j
          BL swap                // call swap
          SUB X20, X20, #1       // j -= 1
          B for2tst              // branch to test of inner loop
exit2:
```

LDR X12,[X10,X20,
LSL #3]

Preserving Registers

- Preserve saved registers:

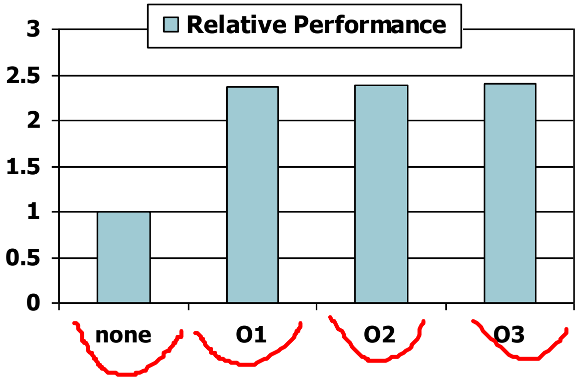
```
SUB SP,SP,#40  // make room on stack for 5 regs
STR LR,[SP,#32]          // save LR on stack
STR X22,[SP,#24]         // save X22 on stack
STR X21,[SP,#16]         // save X21 on stack
STR X20,[SP,#8]          // save X20 on stack
STR X19,[SP,#0]          // save X19 on stack
MOV X21, X0              // copy parameter X0 into X21
MOV X22, X1              // copy parameter X1 into X22
```

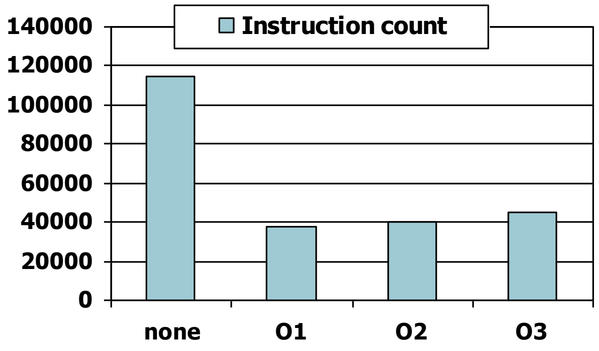
- Restore saved registers:

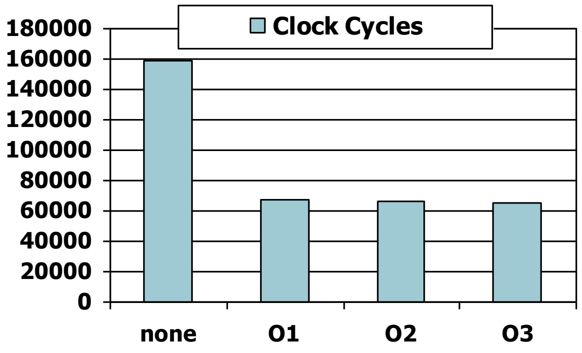
```
exit1: LDUR X19, [SP,#0]  // restore X19 from stack
  LDR X20, [SP,#8]                 // restore X20 from stack
  LDR X21,[SP,#16]                 // restore X21 from stack
  LDR X22,[SP,#24]                 // restore X22 from stack
  LDR X30,[SP,#32]                 // restore LR from stack
  SUB SP,SP,#40           // restore stack pointer
```

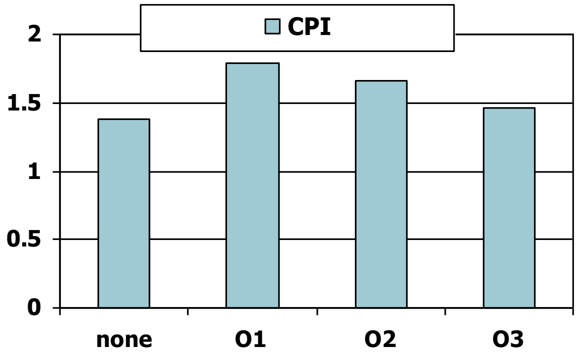
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

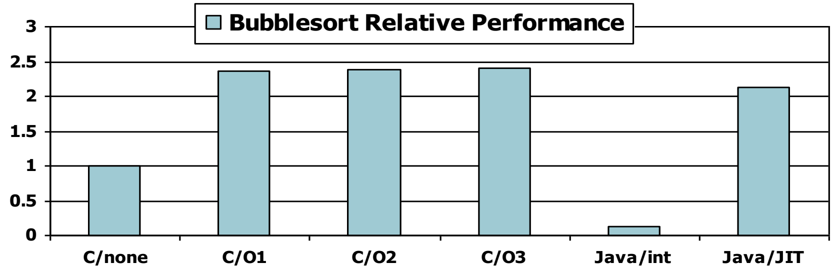


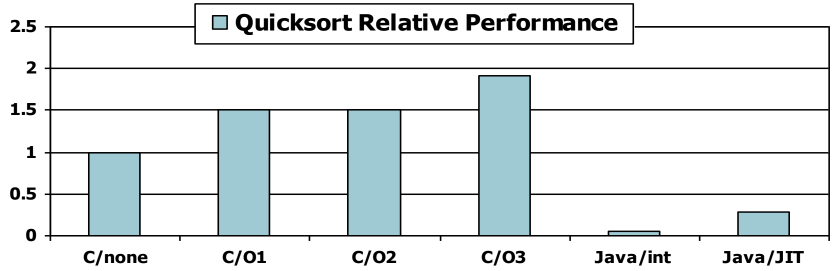


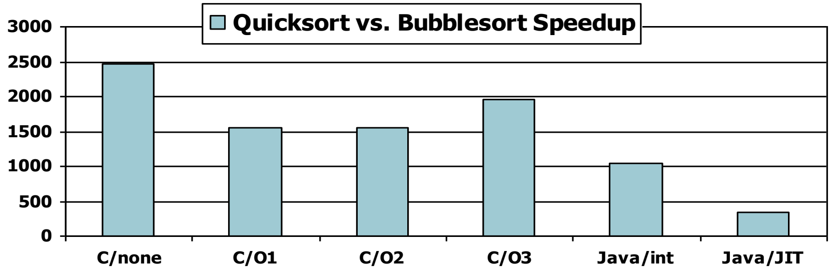




Effect of Language and Algorithm







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!

Arrays vs. Pointers

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

Example: Clearing an Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       MOV X9,XZR     // i = 0
loop1: LSL X10,X9,#3  // X10 = i * 8
       ADD X11,X0,X10 // X11 = address
                      // of array[i]
       STR XZR,[X11,#0]
                      // array[i] = 0
       ADD X9,X9,#1  // i = i + 1
       CMP X9,X1      // compare i to
                      // size
       B.LT loop1     // if (i < size)
                      // go to loop1
```

```
       MOV X9,X0        // p = address of
                        // array[0]
       LSL X10,X1,#3    // X10 = size * 8
       ADD X11,X0,X10   // X11 = address
                        // of array[size]
loop2: STR XZR,0[X9,#0]
                       // Memory[p] = 0
       ADD X9,X9,#8   // p = p + 8
       CMP X9,X11      // compare p to <
                       // &array[size]
       B.LT loop2      // if (p <
                       // &array[size])
                       // go to loop2
```

Comparison of Array vs. Ptr

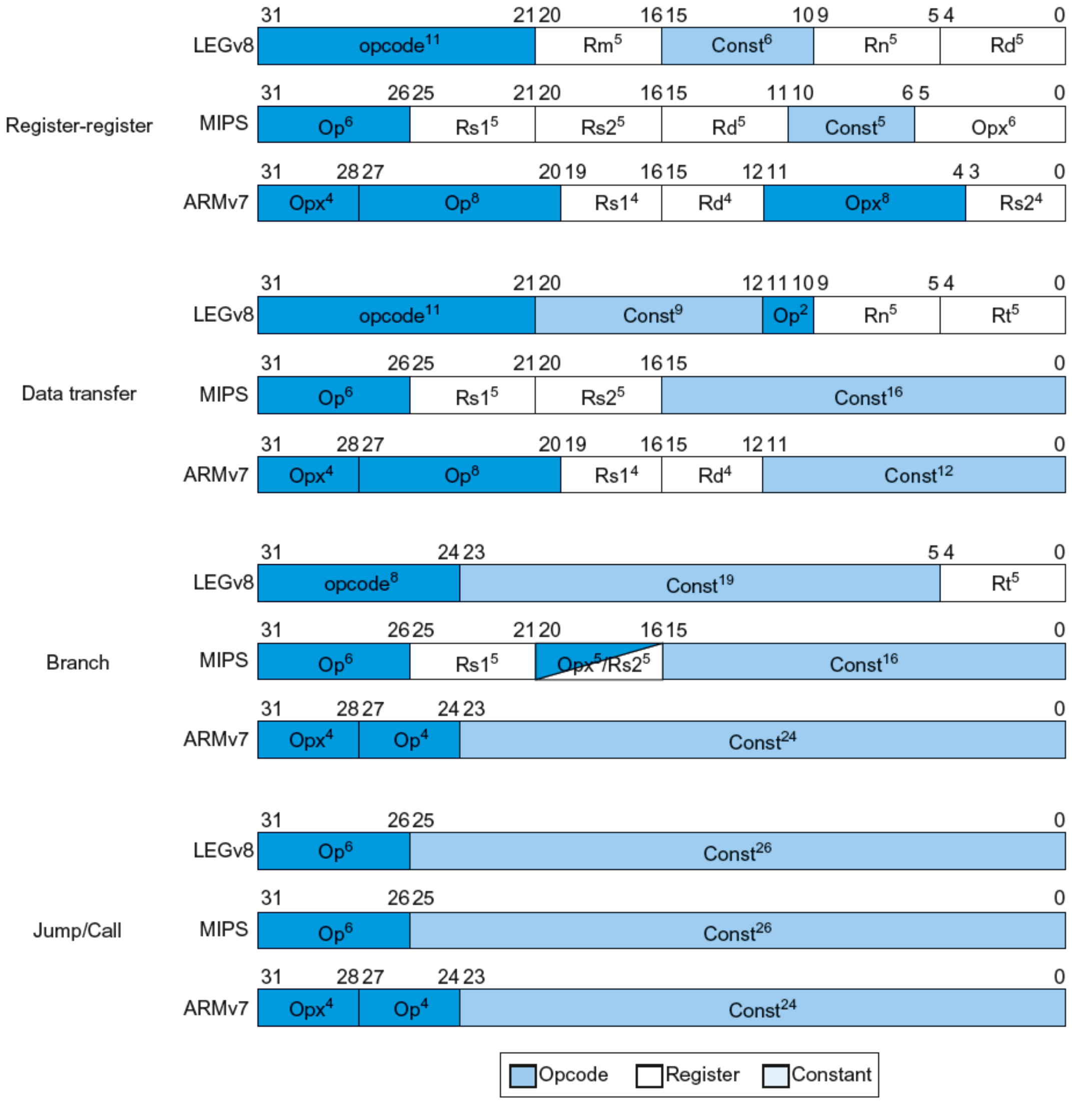
- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

ARM & MIPS Similarities

- 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 Encoding



Register-register

LEGv8: 31 – 21: opcode^{11} | 20 – 16: Rm^{5} | 15 – 10: Const^{6} | 9 – 5: Rn^{5} | 4 – 0: Rd^{5}

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

ARMv7: 31 – 28: Opx^{4} | 27 – 20: Op^{8} | 19 – 16: Rs1^{4} | 15 – 12: Rd^{4} | 11 – 4: Opx^{8} | 3 – 0: Rs2^{4}

Data transfer

LEGv8: 31 – 21: opcode^{11} | 20 – 12: Const^{9} | 11 – 10: Op^{2} | 9 – 5: Rn^{5} | 4 – 0: Rt^{5}

MIPS: 31 – 26: Op^{6} | 25 – 21: Rs1^{5} | 20 – 16: Rs2^{5} | 15 – 0: Const^{16}

ARMv7: 31 – 28: Opx^{4} | 27 – 20: Op^{8} | 19 – 16: Rs1^{4} | 15 – 12: Rd^{4} | 11 – 0: Const^{12}

Branch

LEGv8: 31 – 24: opcode^{8} | 23 – 5: Const^{19} | 4 – 0: Rt^{5}

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

ARMv7: 31 – 28: Opx^{4} | 27 – 24: Op^{4} | 23 – 0: Const^{24}

Jump/Call

LEGv8: 31 – 26: Op^{6} | 25 – 0: Const^{26}

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

ARMv7: 31 – 28: Opx^{4} | 27 – 24: Op^{4} | 23 – 0: Const^{24}

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

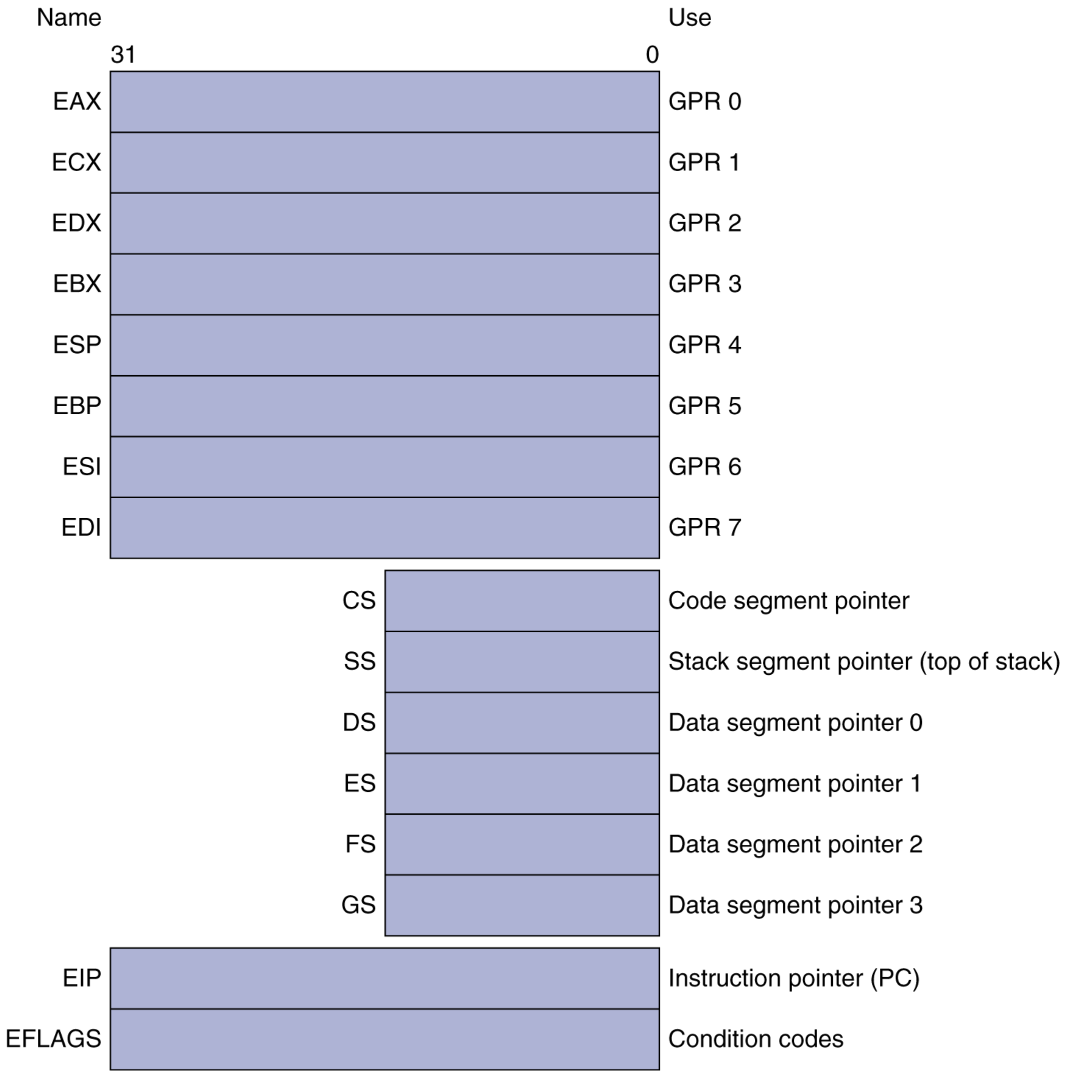
The Intel x86 ISA

- 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
  - 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, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 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

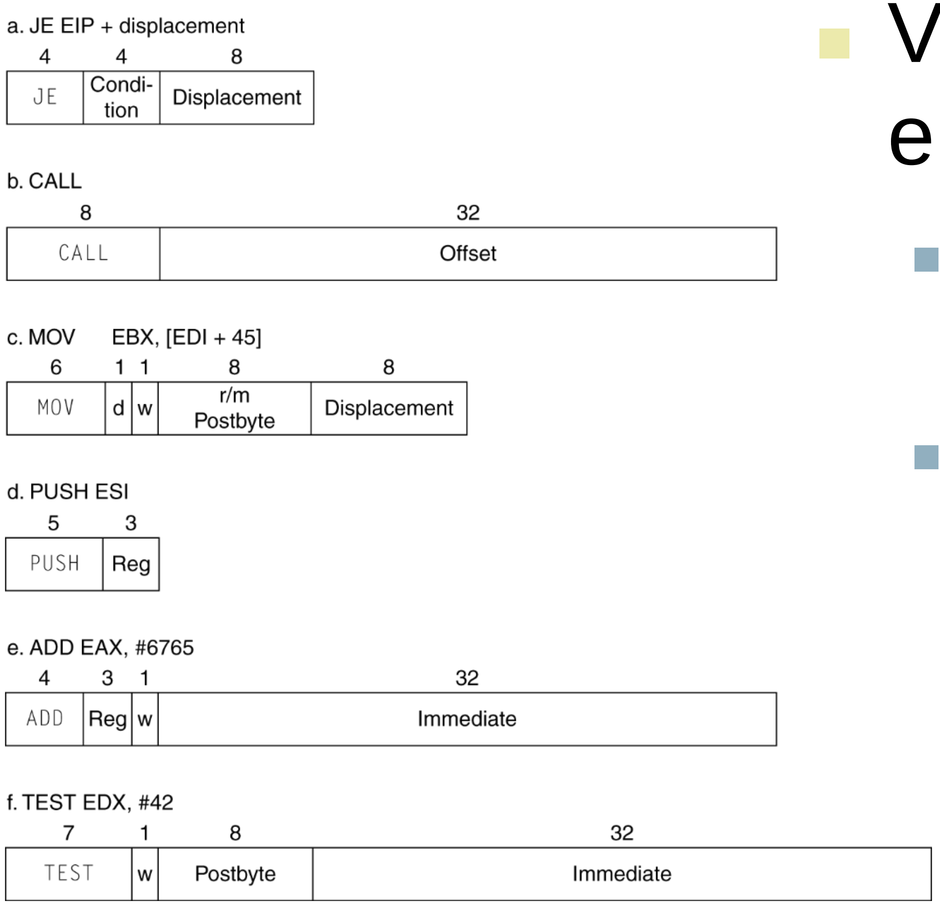
Basic x86 Addressing Modes

- Two operands per instruction

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

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding



a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m 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

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

Implementing IA-32

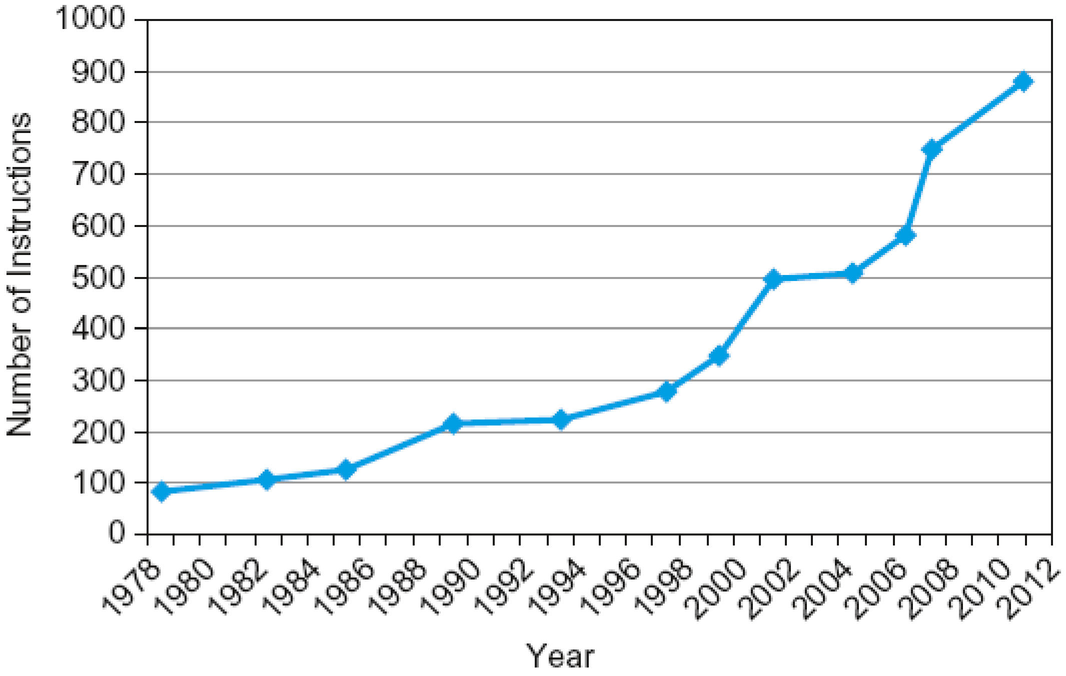
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
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

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
- ARMv8: typical of RISC ISAs
  - c.f. x86