Assembly Language and Computer Organization

Instructions: Language of the Computer

Tuesday, September 13th, 2022

Dr. Robin Pottathuparambil, CSE Department, UNT

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The LEGv8 Instruction Set

- A subset, called LEGv8 (Lessen Extrinsic Garrulity), used as the example throughout the book
- Commercialized by ARM Holdings (www.arm.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See ARM Reference Data tear-out card

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

  ADD a, b, c # a gets b + c
- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Operations

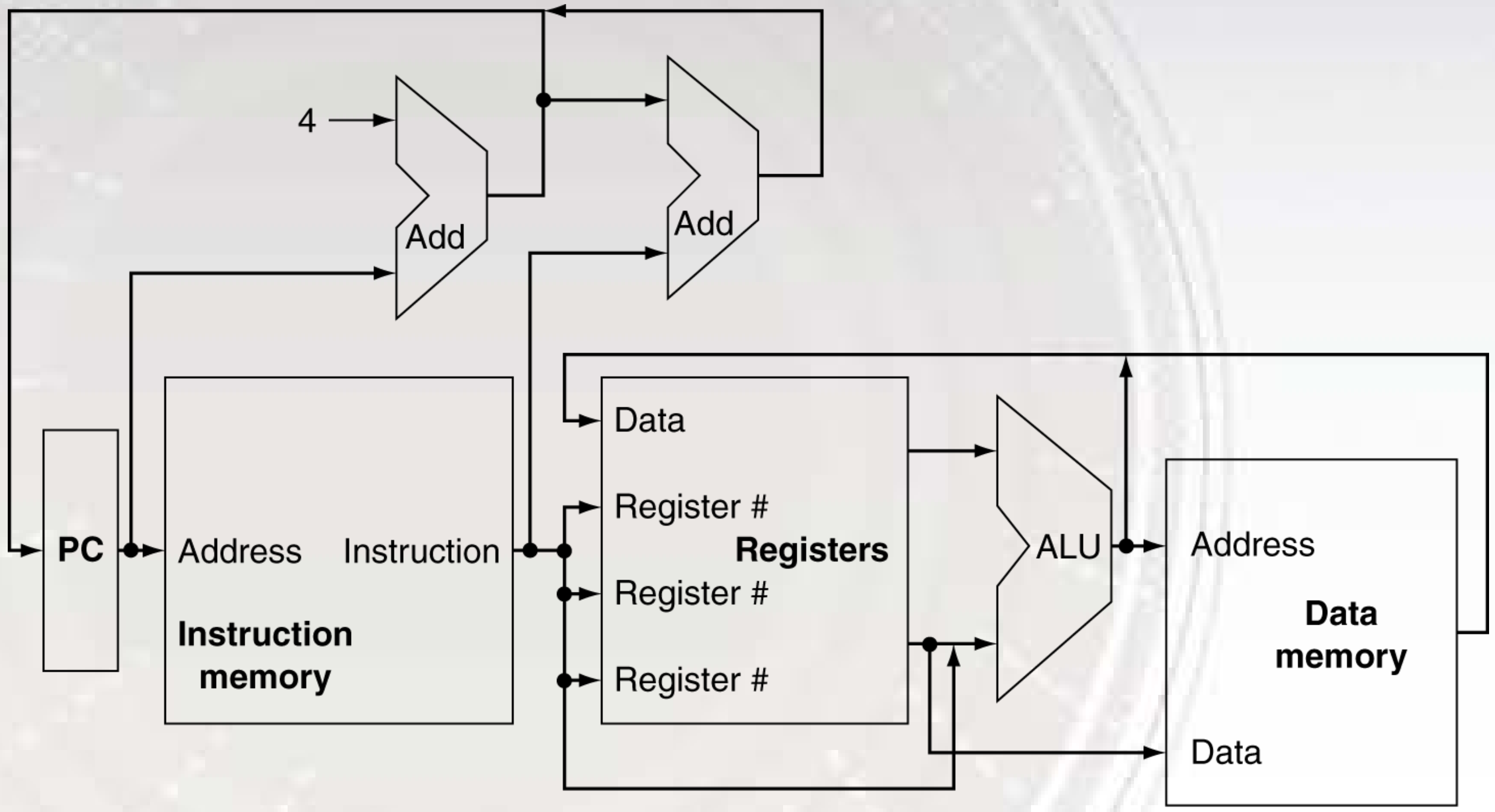
- C code:

```
f = (g + h) - (i + j);
```

- Assembly code (Not the actual code):

```
ADD t0, g, h   # temp t0 = g + h
ADD t1, i, j   # temp t1 = i + j
SUB f, t0, t1  # f = t0 - t1
```

CPU Overview



Register Operands

- Arithmetic instructions use register operands
- LEGv8 has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a “doubleword” or “xword”
    - 31 x 64-bit general purpose registers X0 to X30
  - 32-bit data called a “word”
    - 31 x 32-bit general purpose sub-registers W0 to W30
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Guidelines for LEGv8 Registers

- X0 – X7: procedure arguments/results
- X8: indirect result location register
- X9 – X15: temporaries
- X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register
- X18: platform register for platform independent code; otherwise, a temporary register
- X19 – X27: saved
- X28 (SP): stack pointer
- X29 (FP): frame pointer
- X30 (LR): link register (return address)
- XZR (Register 31): the constant value 0

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  – Assume values for g, h, i, and j is in X20, X21, X22, and X23 respectively and write the result in f in X19

- Compiled LEGv8 code:

```
ADD X9, X20, X21    // Add g and h
ADD X10, X22, X23   // Add i and j
SUB X19, X9, X10    // Sub the results
```

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!

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- LEGv8 does not require words to be aligned in memory, except for instructions and the stack

Memory Operand Example 1

- C code:

```
A[2] = h + A[1];
```

  - Assume h in X21, base address of A in X22, 64-bit integer
- Compiled LEGv8 code:
  - Index 1 requires offset of 8

```
LDUR   X9,[X22,#8]  // U for “unscaled”
ADD    X9,X21,X9    // Add h
STUR   X9,[X22,#16] // Store result in A[2]
```

offset

base register

Shift Operations

- Shift left logical
  - Shift left and fill with 0 bits
  - LSL by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - LSR by *i* bits divides by 2^i (unsigned only)

Memory Operand Example 2

- C code:

```
A[3] = h - A[x];
```

  - Assume h is in X21, base address of A in X22 (64-bit integer), x is in X23
- Compiled LEGv8 code:
  - Index 3 requires offset of 24

```
LSL  X9,X23,#3     // Multiply by 8
ADD  X9,X22,X9     // Add to base address of A
LDUR X9,[X9,#0]    // U for “unscaled”
SUB  X9,X21,X9     // Subtract A[x] from h
STUR X9,[X22,#24] // Store the result in A[3]
```

Immediate Operands

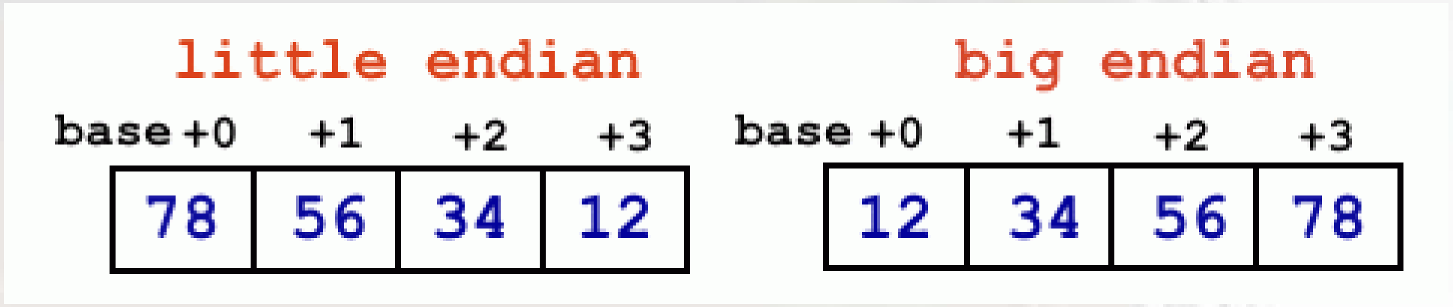
- Constant data specified in an instruction

```
ADDI X22, X22, #4
```

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Little and Big Endian

- int ctr = 0x12345678; // 4 byte integer



Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ .. \ + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + .. + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times 2^{31} + 1\times 2^{30} + \ldots + 1\times 2^2 + 0\times 2^1 + 0\times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2
  - means 1 → 0, 0 → 1

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

- In LEGv8 instruction set
  - LDURSB: sign-extend loaded byte
  - LDURB: zero-extend loaded byte

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

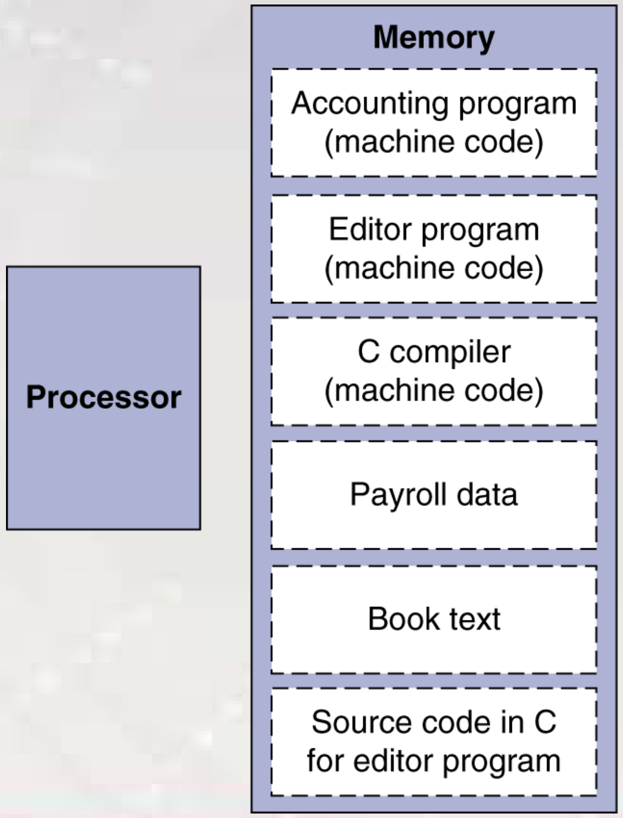
- Example: 0xECA86420
  - 1110 1100 1010 1000 0110 0100 0010 0000

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- LEGv8 instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

Stored Program Computers

The BIG Picture


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

LEGv8 R-format Instructions

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- Instruction fields
  - opcode: operation code
  - Rm: the second register source operand
  - shamt: shift amount (00000 for now)
  - Rn: the first register source operand
  - Rd: the register destination

R-format Example

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

```
ADD X9,X20,X21
```

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}
10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two}$ =

$0x8B150289_{16}$ (Machine Code)

LEGv8 D-format Instructions

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- Load/store instructions
  - Rn: base register
  - address: constant offset from contents of base register (+/- 32 double words or +/- 256)
  - Rt: destination (load) or source (store) register number

- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

D-format Example

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

```
LDUR X9, [X22, #64]
```

1986_{ten}	64_{ten}	0_{ten}	22_{ten}	9_{ten}

11111000010_{two}	001000000_{two}	00_{two}	10110_{two}	01001_{two}

$1111\ 1000\ 0100\ 0100\ 0000\ 0010\ 1100\ 1001_{two}$ =

$0xF84402C9_{16}$ (Machine Code)

LEGv8 I-format Instructions

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- Immediate instructions
  - Rn: source register
  - Rd: destination register

- Immediate field is zero-extended

I-format Example

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

```
ADDI X9, X22, #4
```

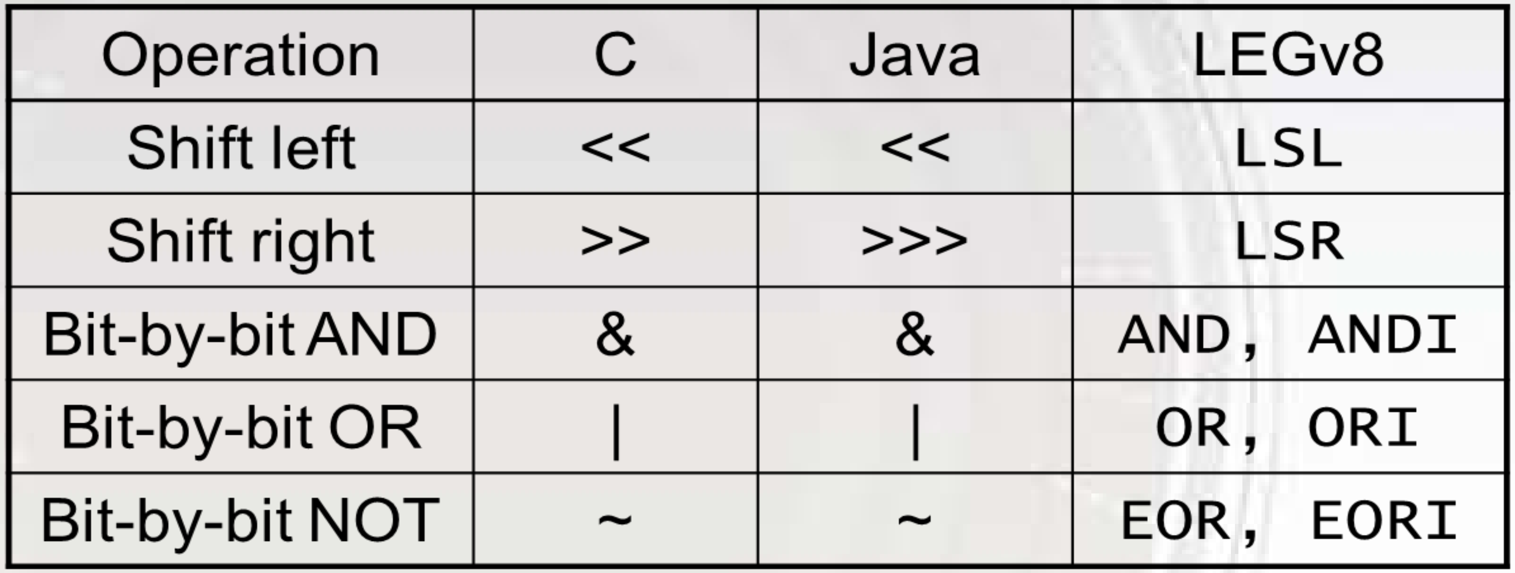
580_{ten}	4_{ten}	22_{ten}	9_{ten}
1001000100_{two}	000000000100_{two}	10110_{two}	01001_{two}

$1001\ 0001\ 0000\ 0000\ 0001\ 0010\ 1100\ 1001_{two}$ =

$0x910012C9_{16}$ (Machine Code)

Logical Operations

- Instructions for bitwise manipulation



Operation	C	Java	LEGv8
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR	|	|	OR, ORI
Bit-by-bit NOT	~	~	EOR, EORI

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

Shift Operations (R-Format)

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - LSL by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - LSR by *i* bits divides by 2^i (unsigned only)

AND Operations

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

AND X9, X10, X11

```
X10  00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X11  00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X9   00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000
```

OR Operations

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

OR X9, X10, X11

X10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

NOT Operations

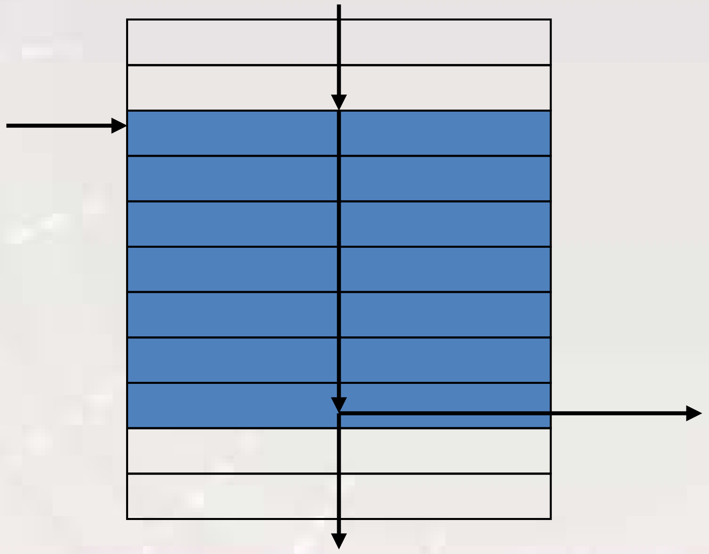
- Differencing operation
  - Set some bits to 1, leave others unchanged

EOR X9, X10, X12 // NOT operation

X10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11000000
X12	11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
X9	11111111	11111111	11111111	11111111	11111111	11111111	11110010	00111111

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Branch Operations

Branch to a labeled instruction if a condition is true
- – Otherwise, continue sequentially

CBZ register, L1
- – if (register == 0) branch to instruction labeled L1;

CBNZ register, L1
- – if (register != 0) branch to instruction labeled L1;

B L1
- – branch unconditionally to instruction labeled L1;

B and CB Format

- B-type
  - `B L1 // go to offset 16`$_{ten}$

5_{ten}	16_{ten} (branch offset)
6 bits	26 bits

- CB-type
  - `CBNZ X19, Exit // go to Exit if X19 != 0`

181_{ten}	22_{ten}(branch offset)	19_{ten}
8 bits	19 bits	5 bits

- Both addresses are PC-relative
  - Address = PC + branch offset (from instruction)

Compiling If Statements

C code:

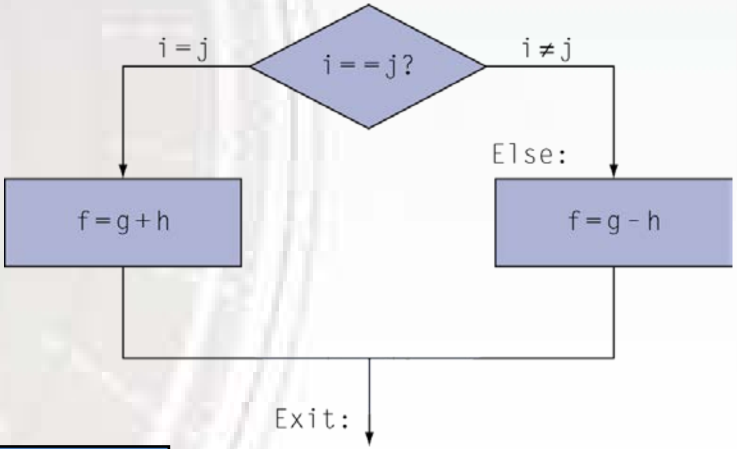
```
if (i == j) f = g + h;
else f = g - h;
```

– Assume f, g, h, i, and j in X19, X20, X21, X22, and X23

Compiled LEGv8 code:

```
      SUB X9,X22,X23
      CBNZ X9,Else
      ADD X19,X20,X21
      B Exit
Else: SUB X19,X20,x21
Exit: …
```

Assembler calculates addresses



A green light to greatness. UNT

Compiling Loop Statements

C snippet:

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

– i in X22, k in X24, address of save in X25, save is 64-bit array

LEGv8 code:

```
Loop: LSL  X10, X22, #3   ; Multiply i*8
      ADD  X10, X10, X25 ; address of save[i]
      LDUR X9, [X10, #0] ; Load save[i]
      SUB  X11, X9, X24  ; check save[i] == k
      CBNZ X11, Exit     ; If false, exit
      ADDI X22, X22, #1  ; else i++
      B    Loop          ; Loop back
Exit: …
```

More Conditional Operations

Condition codes, set from arithmetic instruction with S-suffix (ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS)

- negative (N): result had 1 in MSB
- zero (Z): result was 0
- carry (C): result had carryout from MSB
- overflow (V): result overflowed

Use subtract to set flags, then conditionally branch:

- **B.EQ**
- **B.NE**
- **B.LT** (less than, signed), **B.LO** (less than, unsigned)
- **B.LE** (less than or equal, signed), **B.LS** (less than or equal, unsigned)
- **B.GT** (greater than, signed), **B.HI** (greater than, unsigned)
- **B.GE** (greater than or equal, signed), **B.HS** (greater than or equal, unsigned)

Branch Instruction Design

C Code Snippet:

```
if (a > b) a += 1;
```

– a in X22, b in X23

LEGv8 Code:

```
        SUBS X9, X22, X23 ; Use sub to make comparison
        B.LE Exit         ; Conditional branch
        ADDI X22, X22, #1 ; a++
Exit:
```

For Loop Design

C Code Snippet:

```
for(i=0,i<a,i++) b[i] = a + i;
```

- a is in X22, Base address of array b is in X23, and Data is 64-bit values

For Loop Design

C Code Snippet:

```
for(i=0,i<a,i++) b[i] = a + i;
```

- a is in X22, Base address of array b is in X23, and Data is 64-bit values

LEGv8 Code:

```
        ADDI X9, XZR, #0      ; Assuming i is in temp register X9
  Loop: SUBS X10, X9, X22     ; Check for i < a
        B.GE Exit             ; If false exit else continue
        ADD X10, X22, X9      ; Compute a + i
        LSL X11, X9, #3       ; Multiply i*8 for 64-bits
        ADD X11, X23, X11     ; Compute address for b[i]
        STUR X10, [X11, #0]   ; Store the a + i in b[i]
        ADDI X9, X9, #1       ; i++
        B Loop                ; Loop back
 Exit:
```

Signed vs. Unsigned

Signed comparison

Unsigned comparison

Example

- X22 = 1111 1111 1111 1111 1111 1111 1111 1111
- X23 = 0000 0000 0000 0000 0000 0000 0000 0001
- X22 < X23 # signed
  - –1 < +1
- X22 > X23 # unsigned
  - +4,294,967,295 > +1

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in LEGv8

- Load exclusive register: LDXR
- Store exclusive register: STXR
- To use:
  - Execute LDXR then STXR with same address
  - If there is an intervening change to the address, store fails (communicated with additional output register)
  - Only use register instruction in between

Synchronization in LEGv8

Example 1: atomic swap (to test/set lock variable)

```
again:   LDXR X10,[X20,#0]
         STXR X23,X9,[X20]   // X9 = status
         CBNZ X9, again
         ADD X23,XZR,X10     // X23 = loaded value
```

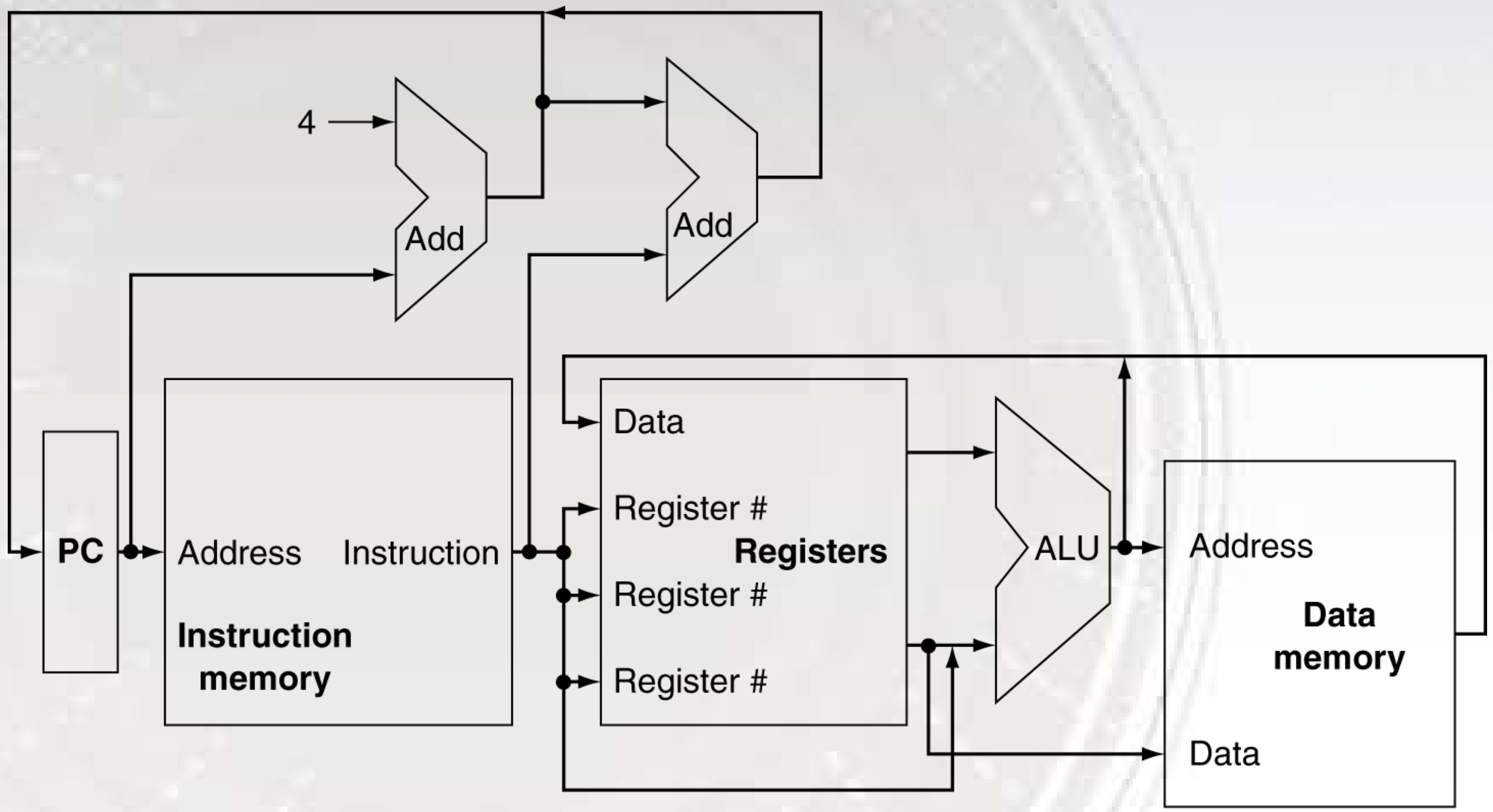
Example 2: lock

```
         ADDI X11,XZR,#1     // copy locked value
again:   LDXR X10,[X20,#0]   // read the lock
         CBNZ X10, again     // check if it is 0 yet
         STXR X11, X9, [X20] // attempt to store
         CBNZ X9, again     // branch if fails
```

– Unlock:

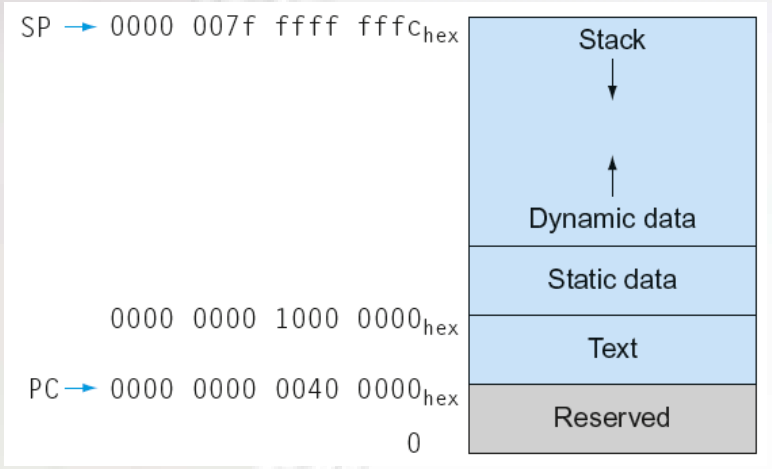
```
         STUR XZR, [X20,#0] // free lock
```

CPU Overview



Memory Layout

- Reserved: Kernel space
- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



A green light to greatness.
UNT

Procedure/Function Example

C code:

```
#include "stdio.h"

long long int a=5, b=4, c;

//Add function
long long int add (long long int x, long long int y)
{
  return (x + y);
}

long long int main(void)  //Main code
{
  c = add(a, b);
  printf("Result: %lld\n", c);
  return 0;
}
```

Procedure Calling

Steps required

1. Place parameters in registers X0 to X7
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 (address in X30)

LEGv8 Registers

- X0 – X7: procedure arguments/results
- X8: indirect result location register
- X9 – X15: temporaries
- X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register
- X18: platform register for platform independent code; otherwise a temporary register
- X19 – X27: saved
- X28 (SP): stack pointer
- X29 (FP): frame pointer
- X30 (LR): link register (return address)
- XZR (register 31): the constant value 0

Procedure Call Instructions

Procedure call: Branch and Link

BL ProcedureLabel

- Address of following instruction put in X30 (LR)
- Jumps to target address

Procedure return: Branch Register

BR LR

- Copies X30 (LR) to program counter
- Can also be used for computed jumps
  - e.g., for case/switch statements

Leaf Procedure Example

C code:

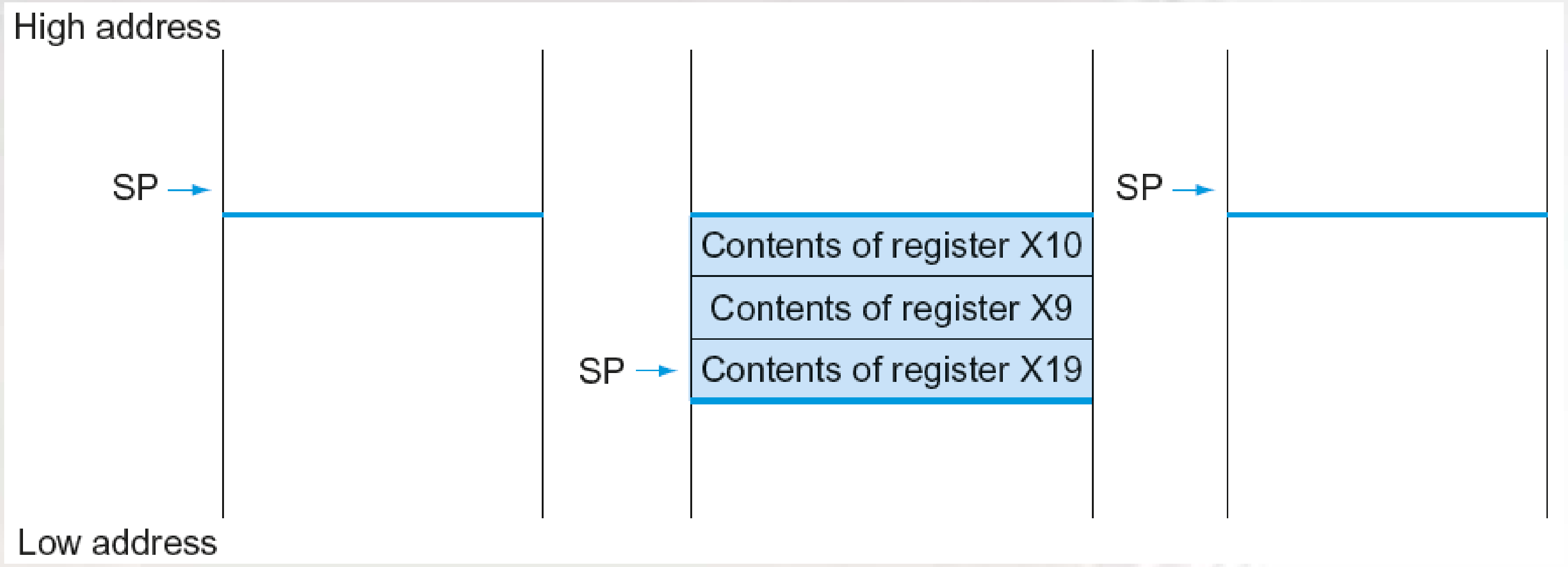
```
long long int leaf_example (long long int g,
long long int h, long long int i, long long
int j)
{
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, h, i, and j in X0, X1, X2, and X3
- Return the result in X0

Leaf Procedure Example

```
leaf:
        SUBI SP,SP,#24            ; Save X10, X9, X19 on stack
        STUR X10,[SP,#16]         ; as its used for computation
        STUR X9,[SP,#8]
        STUR X19,[SP,#0]
        ADD X9,X0,X1              ; X9 = g + h
        ADD X10,X2,X3             ; X10 = i + j
        SUB X19,X9,X10            ; Compute f, X19 = X9 – X10
        ADD X0,X19,XZR            ; Copy f to return register
        LDUR X10,[SP,#16]         ; Restore X10, X9, X19 from stack
        LDUR X9,[SP,#8]
        LDUR X19,[SP,#0]
        ADDI SP,SP,#24
        BR LR                     ; Return to caller
```

Leaf Procedure Example



Register Usage

- X9 to X17: temporary registers
  - Generally not preserved by the callee

- X19 to X28: saved registers
  - If used, the callee saves and restores them

However, if you think you need to preserve the values of the registers (temp or saved), please push it to the stack and pop it after use.

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

Nested Procedure Example

C code:

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

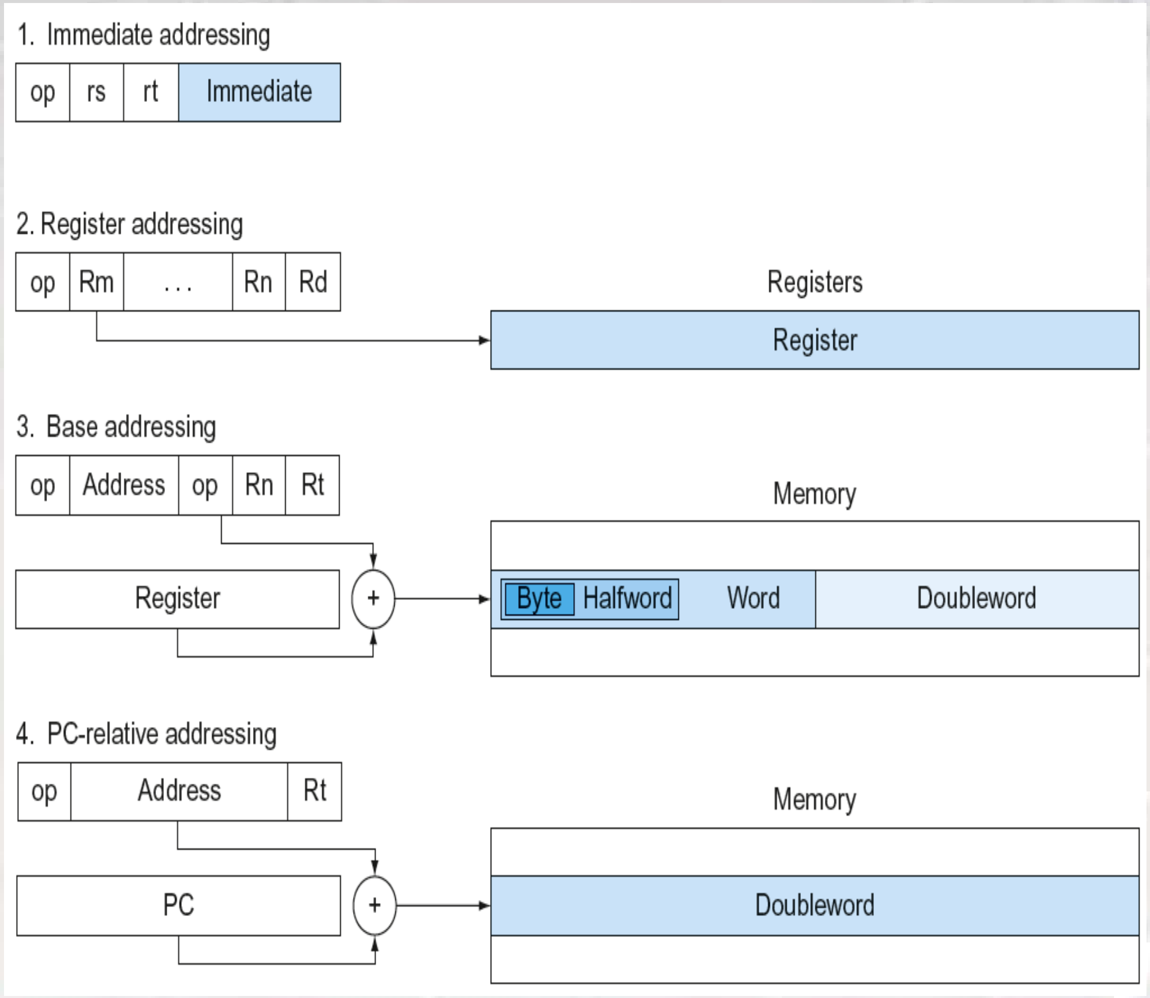
- Argument n in X0
- Result in X1

Nested Procedure Example

- LEGv8 code:

```
fact: SUBI SP,SP,#16          ; Save return address and n on stack
      STUR LR,[SP,#8]         ; Store return address on the stack
      STUR X0,[SP,#0]         ; Store n on the stack
      SUBIS XZR,X0,#1         ; Compare n and 1
      B.GE L1                 ; If n >= 1, go to L1
      ADDI X1,XZR,#1          ; Else, set return value to 1
      ADDI SP,SP,#16          ; Pop stack, don’t bother restoring values
      BR LR                   ; Return
L1:   SUBI X0,X0,#1           ; n = n - 1
      BL fact                 ; Call fact(n-1)
A1:   LDUR X0,[SP,#0]         ; Restore caller’s n
      LDUR LR,[SP,#8]         ; Restore caller’s return address
      ADDI SP,SP,#16          ; Pop stack
      MUL X1,X0,X1            ; Return n * fact(n-1)
      BR LR                   ; Return
```

Addressing Mode Summary



ADDI X0, X1, #2

ADD X0, X1, X2

LDUR X0, [X1, #0]

CBZ X1, L1

Examples for Addressing modes

32-bit Constants (IW Format)

- Most constants are small
  - – 12-bit immediate is sufficient
- For the occasional 32-bit constant

  MOVZ: move wide with zeros

  MOVK: move wide with keep
- Use with flexible second operand (shift)

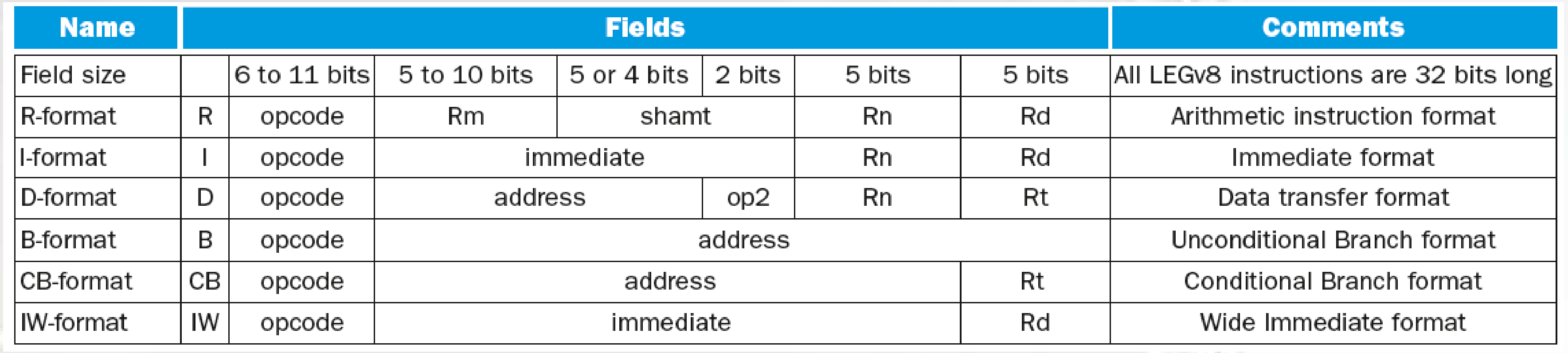
```
MOVZ X9,255,LSL 16
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 0000 0000

```
MOVK X9,255,LSL 0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 1111 1111

LEGv8 Encoding Summary



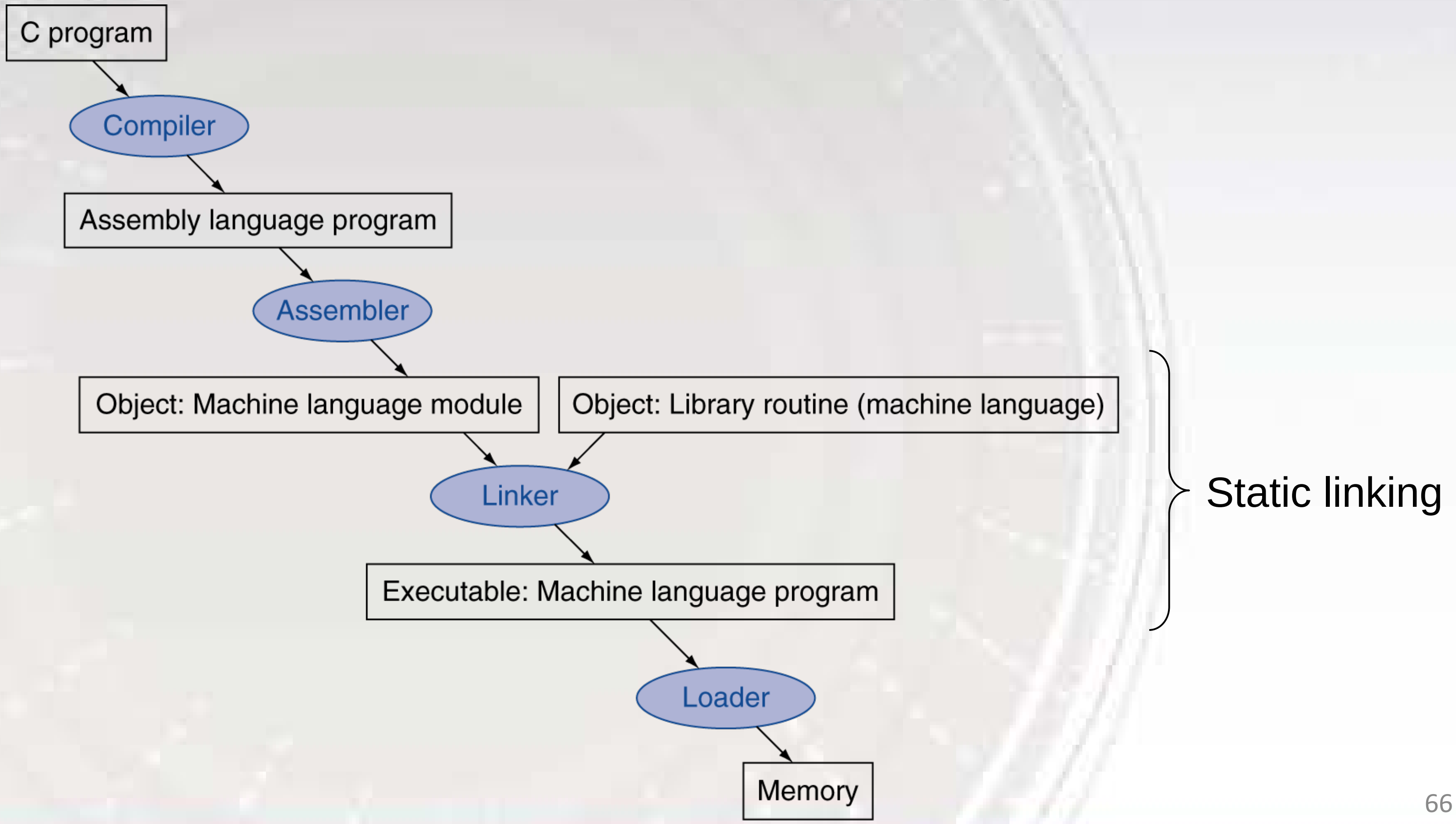
Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination

```
MOV X9, X10       →   ADD X9, X10, XZR
CMP X9, X10       →   SUBS XZR, X9, X10
CMPI X9, #2       →   SUBIS XZR, X9, #2
```

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
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 internal and external refs

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

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

LEGv8 byte/halfword load/store

- Load byte:
  - LDURB Rt, [Rn, offset]
  - Sign extend to 32 bits/64 bits in Rt
- Store byte:
  - STURB Rt, [Rn, offset]
  - Store just rightmost byte
- Load halfword:
  - LDURH Rt, [Rn, offset]
  - Sign extend to 32 bits/64 bits in Rt
- Store halfword:
  - STURH Rt, [Rn, offset]
  - Store just rightmost halfword

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{
  unsigned long long int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Base addresses of x, y in X0, X1
  - i in X19

String Copy Example

- LEGv8 code:

```
strcpy:
        SUBI SP,SP,#24       // Create space on stack
        STUR X9,[SP,#16]     // Push X9
        STUR X10,[SP,#8]     // Push X10
        STUR X19,[SP,#0]     // Push X19
        ADDI X19,XZR,#0      // i=0
L1:     ADD X9,X19,X1        // X9 = addr of y[i]
        LDURB X10,[X9,#0]    // X10 = y[i]
        ADD X9,X19,X0        // X9 = addr of x[i]
        STURB X10,[X9,#0]    // x[i] = y[i]
        CBZ X10,L2           // if y[i] == 0 then exit
        ADDI X19,X19,#1      // i = i + 1
        B L1                 // next iteration of loop
L2:     LDUR X9,[SP,#16]     // restore saved X9
        LDUR X10,[SP,#8]     // restore saved X10
        LDUR X19,[SP,#0]     // restore saved X19
        ADDI SP,SP,#24       // pop 3 item from stack
        BR LR                // and return
```

The Sort Procedure in C

- Illustrates use of assembly instructions for a C bubble sort function (time complexity - O(n^2))
- The below function performs **ascending** order sort

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

Base address of v in X21, n in X22, i in X19, and j in X20

Swap procedure

- Swap procedure

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

Base address of 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]
      LDUR X9,[X10,#0]   // X9 = v[k]
      LDUR X11,[X10,#8]  // X11 = v[k+1]
      STUR X11,[X10,#0]  // v[k] = X11 (v[k+1])
      STUR X9,[X10,#8]   // v[k+1] = X9 (v[k])
      BR LR              // return to calling routine
```

Base address of v in X0, k in X1, temp in X9

Assume X9, X10, and X11 are free to be used by the swap function.

The Outer Loop

Outer loop:

- for (i = 0; i <n; i++) {

```
          MOV X19,XZR        // i = 0
for1tst:
          CMP X19, X22       // compare X19 to X22 (i to n)
          B.GE exit1         // go to exit1 if X19 ≥ X1 (i≥n)

          (body of inner for-loop)

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

The Inner Loop

Inner loop: (Base address of v in X21, n in X22, i in X19, and j in X20)

- for (j = i – 1; ((j >= 0) && (v[j] > v[j + 1])); j--) {

```
          SUBI 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, X21, X10    // reg X11 = v + (j * 8)
         LDUR X12, [X11,#0]   // reg X12 = v[j]
         LDUR 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
         SUBI X20, X20, #1    // j--
         B for2tst            // branch to test of inner loop
exit2:
```

Preserving Registers (Sort)

Preserve saved registers: (*Just preserve saved registers not the temp registers for this example*)

```
        SUBI SP,SP,#40    // make room on stack for 5 regs
        STUR LR,[SP,#32]  // save LR on stack  (to return to caller)
        STUR X22,[SP,#24] // save X22 on stack (used for n)
        STUR X21,[SP,#16] // save X21 on stack (used for v)
        STUR X20,[SP,#8]  // save X20 on stack (j)
        STUR X19,[SP,#0]  // save X19 on stack (i)
        MOV X21, X0        // copy parameter X0 into X21 before start
        MOV X22, X1        // copy parameter X1 into X22 before start
```

Restore saved registers:

```
exit1: LDUR X19, [SP,#0] // restore X19 from stack
       LDUR X20, [SP,#8] // restore X20 from stack
       LDUR X21,[SP,#16] // restore X21 from stack
       LDUR X22,[SP,#24] // restore X22 from stack
       LDUR LR,[SP,#32]   // restore LR from stack
       ADDI SP,SP,#40    // restore stack pointer
       BR LR             // return to caller
```

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(int64_t array[], int64_t size) {
  long long int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int64_t *array, int64_t size) {
  long long 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]
       STUR XZR,[X11,#0]
                      // array[i] = 0
       ADDI X9,X9,#1  // i = i + 1
       CMP X9,X1      // compare i to
                      // size
       B.LT loop1     // if (i < size)
                      // go to loop1

X0 – base address of array
X1 – size of the array
```

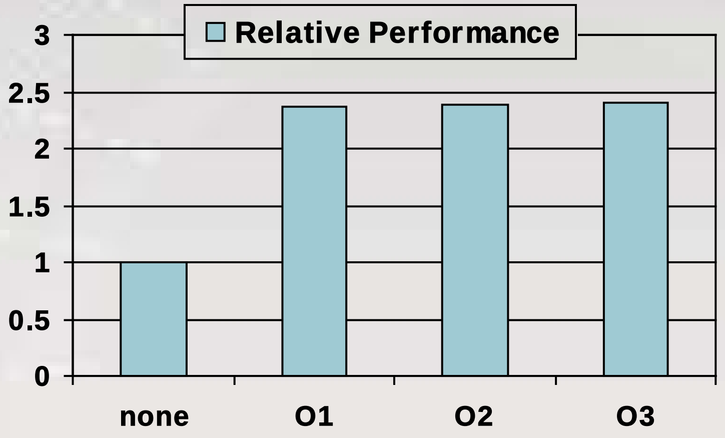
```
       MOV X9,X0       // p = address of
                       // array[0]
       LSL X10,X1,#3   // X10 = size * 8
       ADD X11,X0,X10  // X11 = address
                       // of array[size]
loop2: STUR XZR,[X9,#0]
                      // Memory[p] = 0
       ADDI X9,X9,#8  // p = p + 8
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

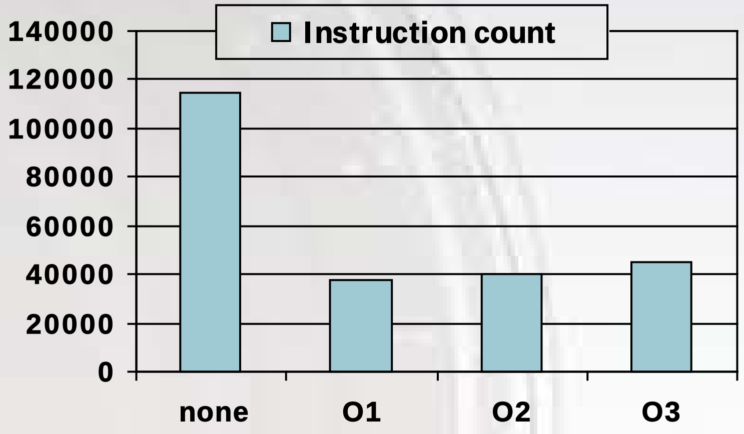
Comparison of Array vs. Ptr

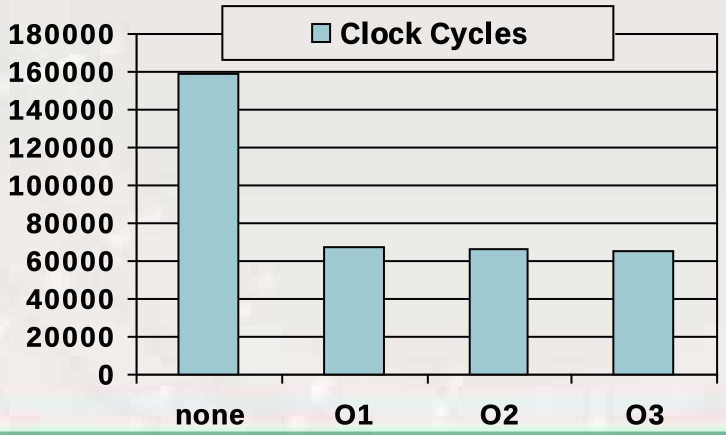
- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and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

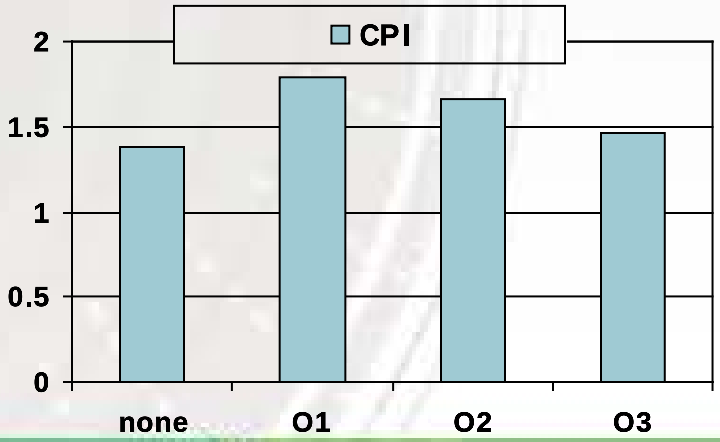
Effect of Compiler Optimization

Bubble sort Compiled with gcc for Pentium 4 under Linux

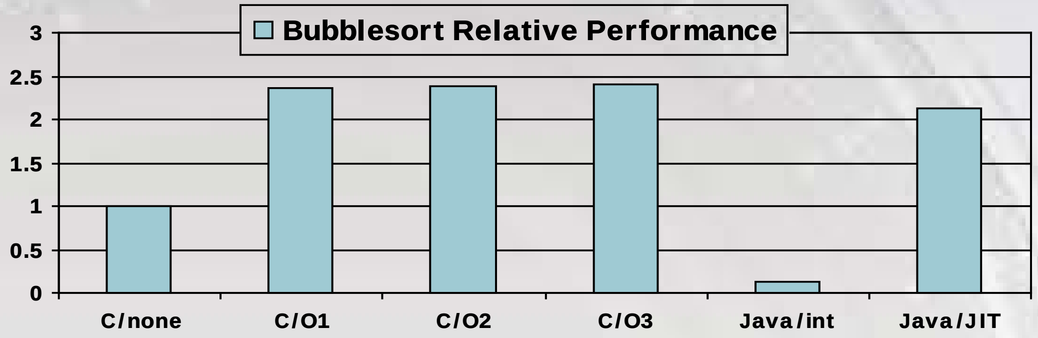


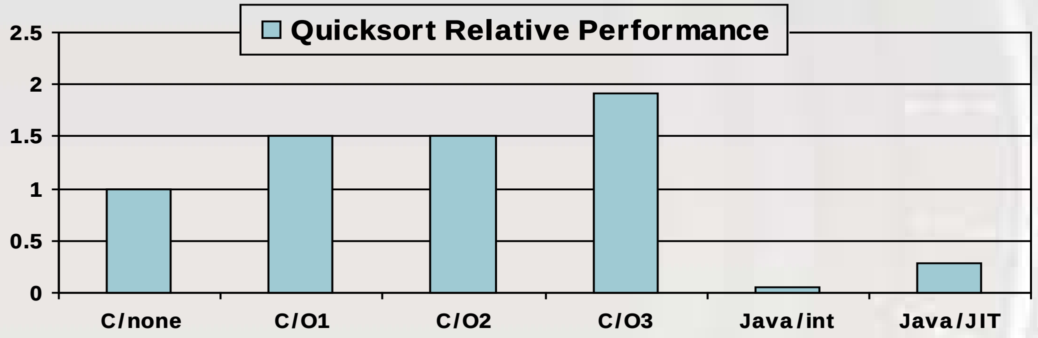


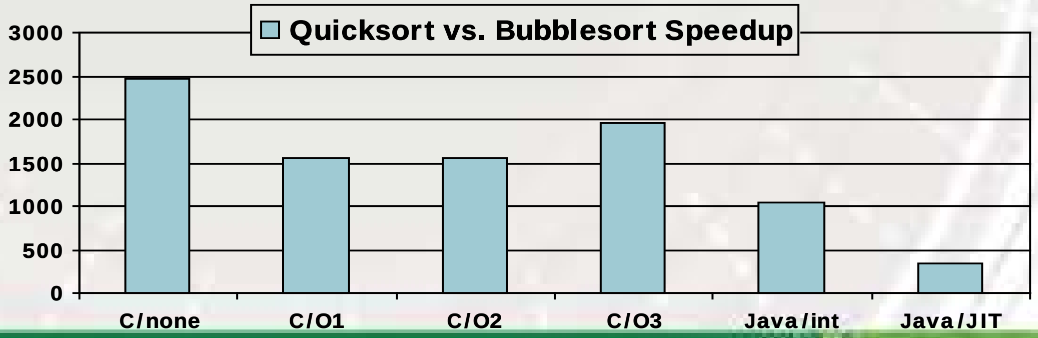




Effect of Language and Algorithm





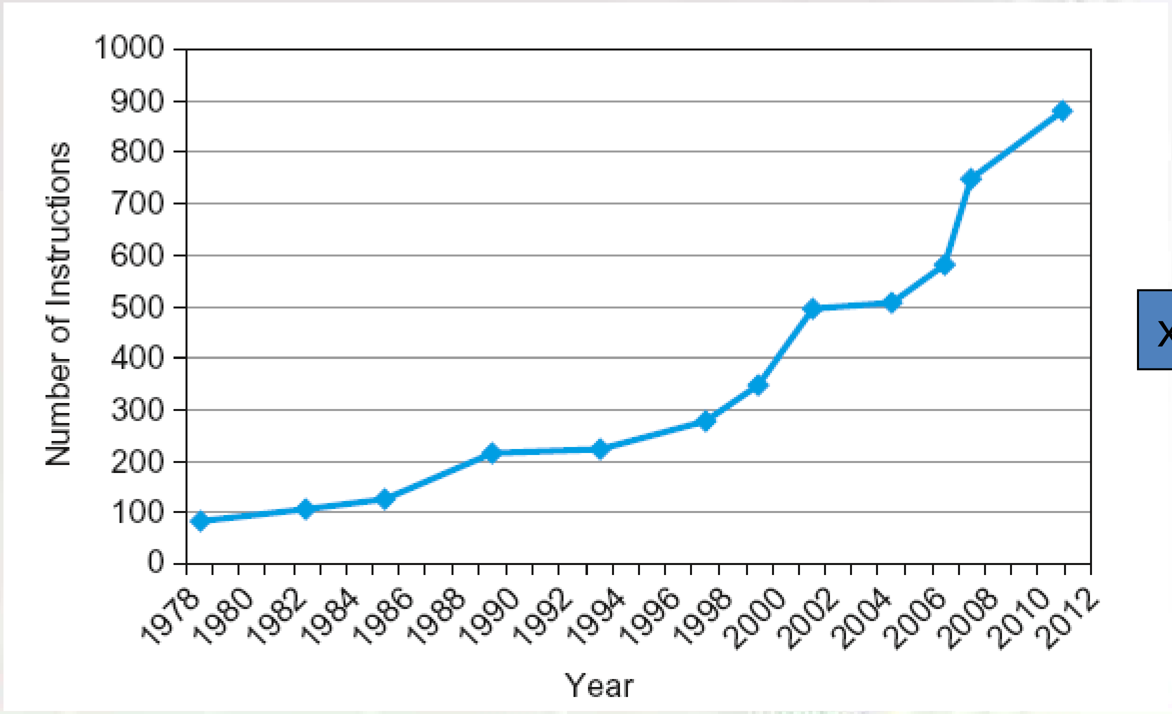


Fallacies

- Powerful instruction $\Rightarrow$ 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 $\Rightarrow$ more errors and less productivity

Fallacies

- Backward compatibility $\Rightarrow$ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 8 for double word, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  2. Good design demands good compromises
- Layers of software/hardware
  – Compiler, assembler, hardware
- LEGv8 Instructions and formats.