Chapter 2

Instructions: Language of the Computer

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 ARM Instruction Set

- Used as the example in chapters 2 and 3
- Most popular 32-bit instruction set in the world (www.arm.com)
- 4 Billion shipped in 2008
- Large share of embedded core market
  - Applications include mobile phones, consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern RISC ISAs
  - See ARM Assembler instructions, their encoding and instruction cycle timings in appendixes B1,B2 and B3 (CD-ROM)

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

Arithmetic Example

- C code:

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

- Compiled ARM code:

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

Register Operands

- Arithmetic instructions use register operands
- ARM has a 16 × 32-bit register file
  - Use for frequently accessed data
  - Registers numbered 0 to 15 (r0 to r15)
  - 32-bit data called a “word”
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

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

  - *f, …, j in registers r0, …,r4*
  - *r5 and r6 are temporary registers*

- Compiled ARM code:

```
ADD r5,r0,r1 ;register r5 contains g + h
ADD r6,r2,r3 ;register r6 contains i + j
SUB r4,r5,r6 ;r4 gets r5-r6
```

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
- Words are aligned in memory
  - Address must be a multiple of 4
- ARM is Little Endian
  - Least-significant byte at least address
  - *c.f.* Big Endian: Most-significant byte at least address of a word

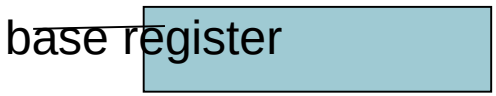
Memory Operand Example 1

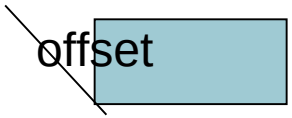
- C code:

```
g = h + A[8];
```

  - *g in r1, h in r2, base address of A in r3*
  - *r5 is temporary register*
- Compiled ARM code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
LDR  r5,[r3,#32] ; reg r5 gets A[8]
ADD r1, r2, r5 ; g = h + A[8]
```





Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - *h in r2, base address of A in r3*
  - *r5 is temporary register*
- Compiled ARM code:
  - Index 8 requires offset of 32

```
LDR  r5,[r3,#32] ; reg r5 gets A[8]
ADD  r5, r2, r5  ; reg r5 gets h+A[8]
STR  r5,[r3,#48] ; Stores h+A[8]into A[12]
```

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!

Immediate Operands

- Constant data specified in an instruction

```
ADD r3,r3,#4 ; r3 = r3 + 4
```

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

Unsigned Binary Integers

- Given an n-bit number

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

- 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

2s-Complement Signed Integers

- Given an n-bit number

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

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- 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

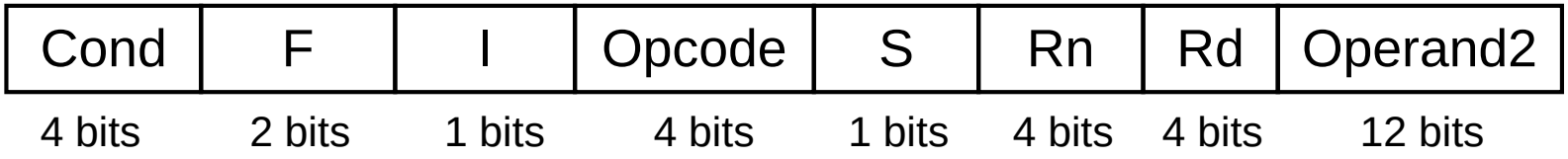
Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In ARM instruction set
  - `LDRSB`, `LDRSH`: extend loaded byte/halfword
- 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

Representing Instructions

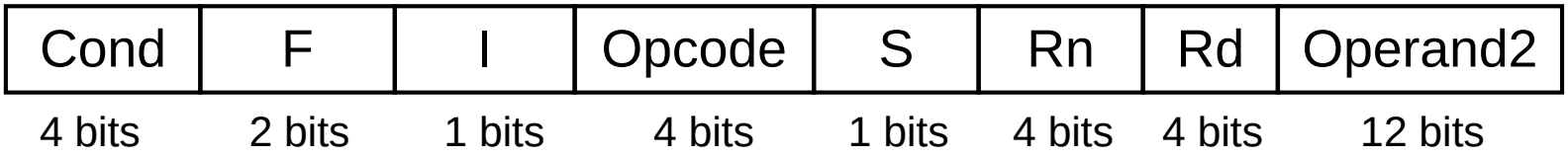
- Instructions are encoded in binary
  - Called machine code
- ARM instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers – r0 to r15

ARM Data Processing (DP) Instructions

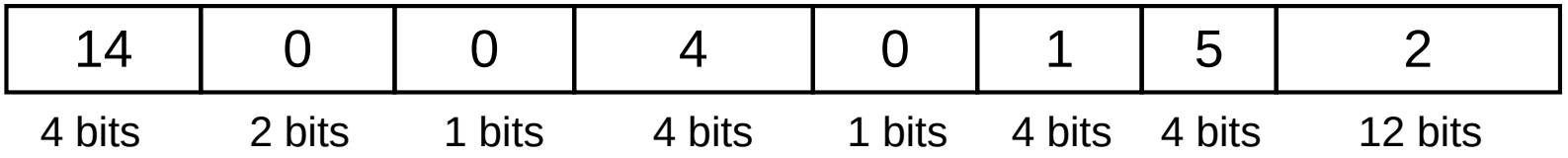


- Instruction fields
  - Opcode : Basic operation of the instruction
  - Rd: The destination register operand
  - Rn: The first register source operand
  - Operand2: The second source operand
  - I:Immediate. If I is 0, the second source operand is a register, else the second source is a 12-bit immediate.
  - S: Set Condition Code
  - Cond: Condition
  - F: Instruction Format.

DP Instruction Example




```
ADD r5,r1,r2 ; r5 = r1 + r2
```



$$11100000100000010101000000000010_2$$

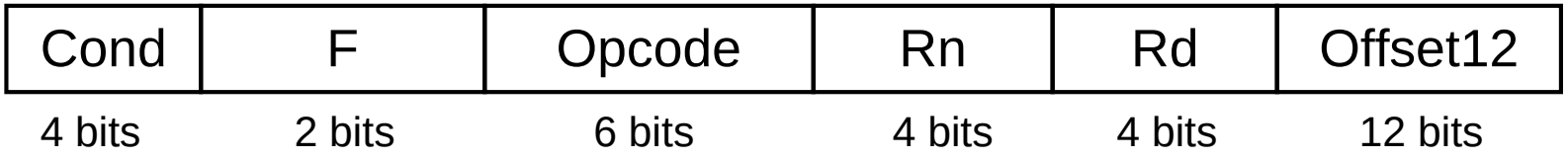
Hexadecimal

- 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: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

ARM Data Transfer (DT) Instruction

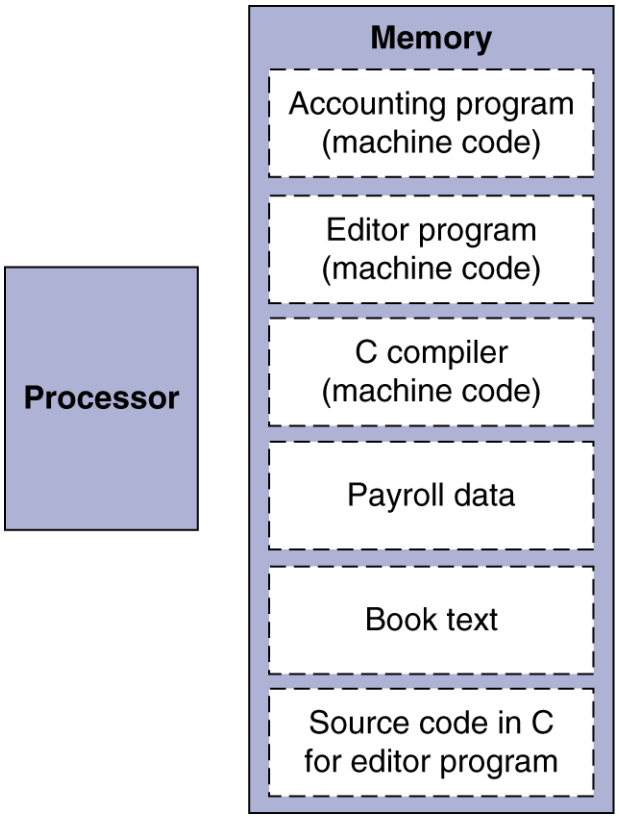


LDR r5, [r3, #32] ; Temporary reg r5 gets A[8]

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Stored Program Computers

The BIG Picture



- 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

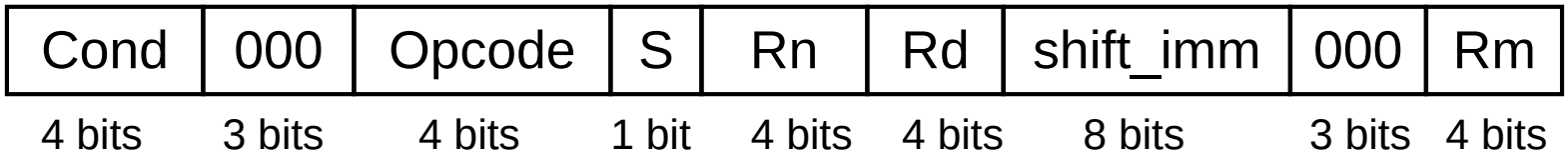
Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	ARM
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR
Bitwise AND	&	&	AND
Bitwise OR	|	|	ORR
Bitwise NOT	~	~	MVN

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

Shift Operations

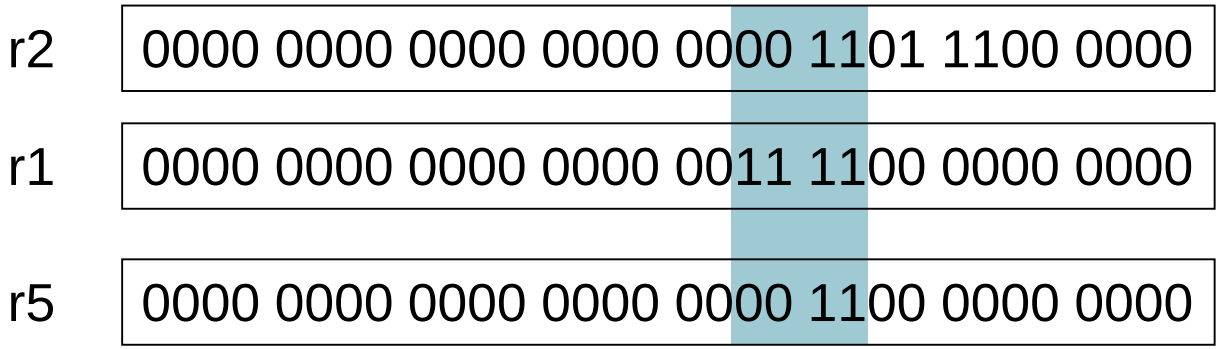


- shift_imm: how many positions to shift
- Logical shift left (LSL)
  - Shift left and fill with 0 bits
  - LSL by *i* bits multiplies by 2^i
    - *rm, LSL #<shift_imm>*
    - *ADD r5,r1,r2 LSL #2 ; r5 = r1 + (r2 << 2)*
- Logical shift right(LSR)
  - Shift right and fill with 0 bits
  - LSR by *i* bits divides by 2^i (unsigned only)
    - *rm, LSR #<shift_imm>*
    - *MOV r6,r5, LSR # 4 ; r6 = r5 >> 4*

AND Operations

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

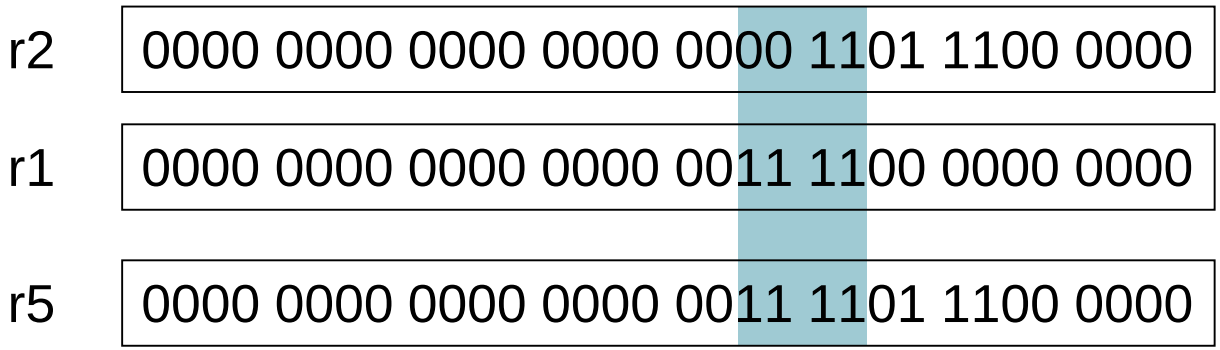
```
AND r5, r1, r2 ; reg r5 = reg r1 & reg r2
```



OR Operations

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

```
ORR r5, r1, r2  ; reg r5 = reg r1 | reg r2
```



NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- ARM has Move Not (MVN)

```
MVN r5, r1 ;  reg r5 = ~ reg r1
```

r1	0000 0000 0000 0000 0011 1100 0000 0000
r5	1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `CMP reg1,reg2`
- `BEQ L1`
  - if (reg1 == reg2) branch to instruction labeled L1;
- `CMP reg1,reg2`
- `BNE L1`
  - if (reg1 != reg2) branch to instruction labeled L1;
- `B exit ; go to exit`
  - unconditional jump to instruction labeled exit

ARM instruction format summary

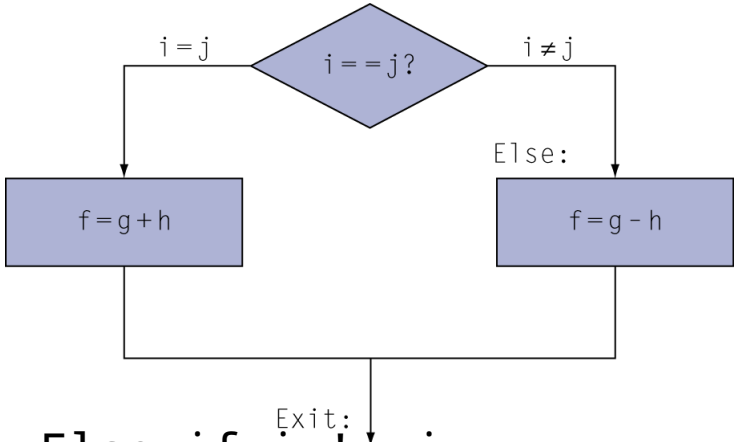
Name	Format	Example								Comments
Field size		4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits	All ARM instructions are 32 bits long
DP format	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	Arithmetic instruction format
DT format	DT	Cond	F	Opcode			Rn	Rd	Offset12	Data transfer format
Field size		4 bits	2 bits	2 bits	24 bits					
BR format	BR	Cond	F	Opcode	signed_immed_24					B and BL instructions

Compiling If Statements L

- C code:

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

  - *f, g, … in r0, r1,..r4*



- Compiled ARM code:

```
        CMP r3,r4
        BNE Else                ; go to Else if i != j

        ADD r1,r1,r2            ; f = g + h (skipped if i != j)
        B Exit
Else :                          ; f = g - h (skipped if i = j)
        SUB r0,r1,r2
Exit:
```

Assembler calculates addresses

Compiling Loop StatementsL

- C code:

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

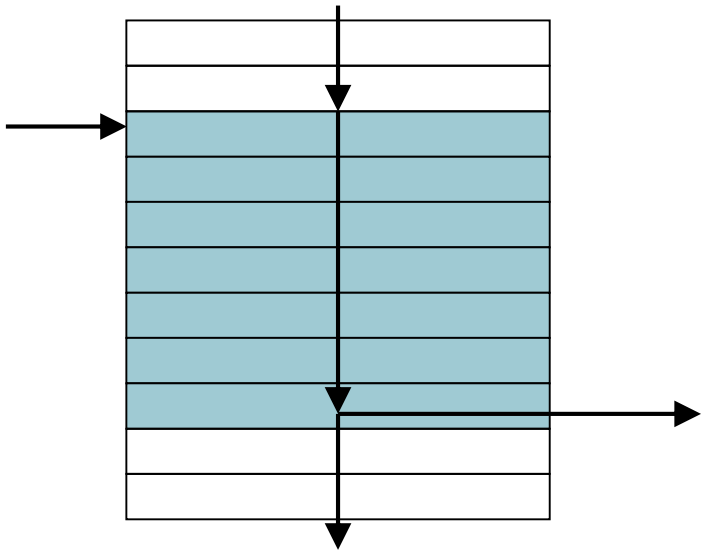
  - *i in r3, k in r5, base address of save in r6*

- Compiled ARM code:

```
Loop:  ADD r12,r6, r3, LSL # 2  ; r12 = address of save[i]
       LDR r0,[r12,#0]          ; Temp reg r0 = save[i]
       CMP r0,r5
       BNE Exit                 ; go to Exit if save[i] ≠ k
       ADD r3,r3,#1             ; i = i + 1
       B Loop                   ; go to Loop
Exit:
```

Basic BlocksL

- 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

Signed vs. Unsigned

- Signed comparison: `BGE,BLT,BGT,BLE`
- Unsigned comparison: `BHS,BLO,BHI,BLS`
- Example
  - r0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - r1 = 0000 0000 0000 0000 0000 0000 0000 0001
    - `CMP r0,r1`
    - `BLO L1 ; unsigned branch`
      - Branch not taken since $4{,}294{,}967{,}295_{ten} > 1_{ten}$
    - `BLT L2  ; signed branch`
      - Branch taken to L2 since $-1_{ten} < 1_{ten}$.

Procedure CallingL

- 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

ARM register conventions

Name	Register number	Usage	Preserved on call?
a1-a2	0–1	Argument / return result / scratch register	no
a3-a4	2–3	Argument / scratch register	no
v1-v8	4–11	Variables for local routine	yes
ip	12	Intra-procedure-call scratch register	no
sp	13	Stack pointer	yes
lr	14	Link Register (Return address)	yes
pc	15	Program Counter	n.a.

Procedure Call InstructionsL

- Procedure call: Branch and link

  `BL ProcedureAddress`

  - Address of following instruction put in ***lr***
  - Jumps to target address
- Procedure return:

  `MOV pc,lr`

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

Leaf Procedure ExampleL

- C code:

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

  - *Arguments g, …, j in r0, …, r3*
  - *f in r4 (hence, need to save r4 on stack)*
  - *Result in r0*

Leaf Procedure ExampleL

- ARM code:

```
leaf_example:
    SUB sp, sp, #12
    STR r6,[sp,#8]
    STR r5,[sp,#4]
    STR r4,[sp,#0]
    ADD r5,r0,r1
    ADD r6,r2,r3
    SUB r4,r5,r6
    MOV r0,r4
    LDR r4, [sp,#0]
    LDR r5, [sp,#4]
    LDR r6, [sp,#8]
    ADD sp,sp,#12
    MOV pc,lr
```

Make room for 3 items

Save r4,r5,r6 on stack

r5 = (g+h), r6= (i+j)
Result in r4

Result moved to return value register r0.

Restore r4,r5,r6

Return

Non-Leaf ProceduresL

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

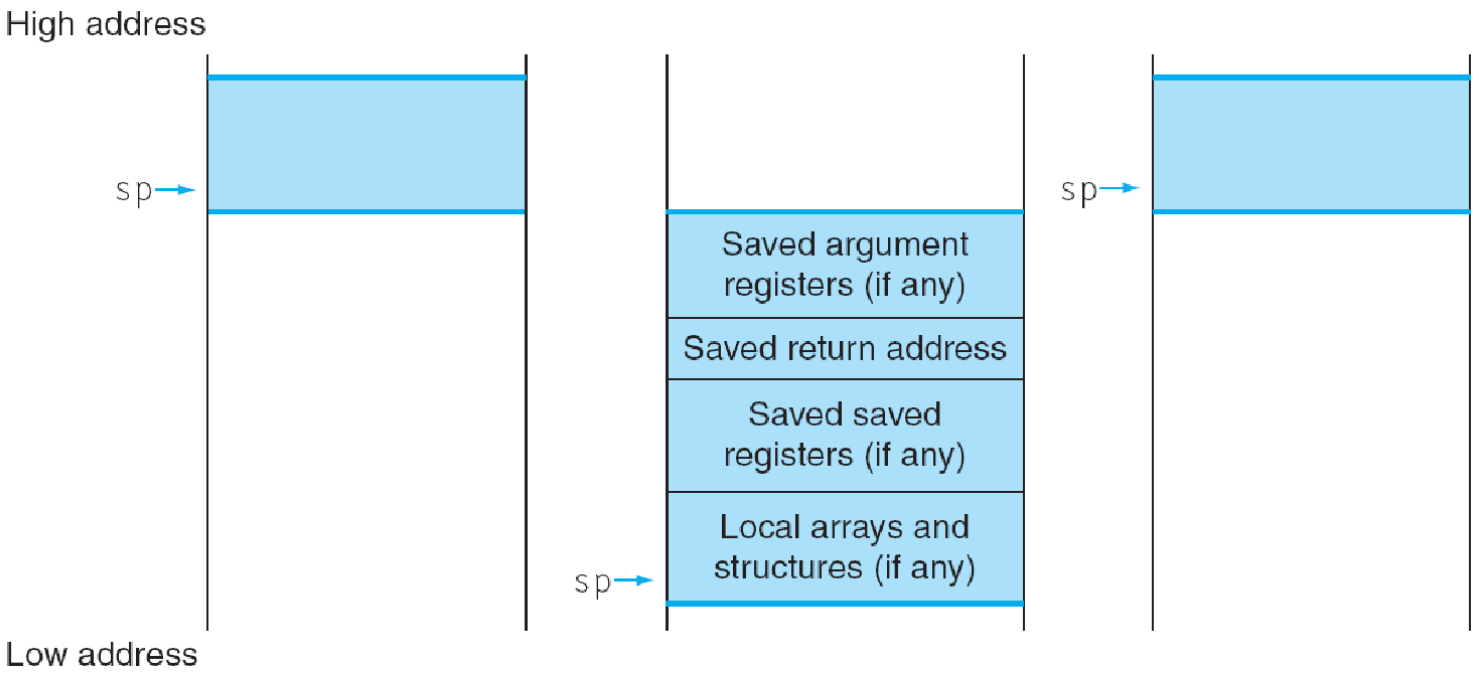
  - *Argument n in register r0*
  - *Result in register r0*

Non-Leaf Procedure Example

- ARM code:

```
fact:
    SUB sp,sp,#8        ; Adjust stack for 2 items
    STR lr,[sp,#4]      ; Save return address
    STR r0,[sp,#0]      ; Save argument n
    CMP r0,#1           ; compare n to 1
    BGE L1
    MOV r0,#1           ; if so, result is 1
    ADD sp,sp,#8        ; Pop 2 items from stack
    MOV pc,lr           ; Return to caller
L1: SUB r0,r0,#1        ; else decrement n
    BL  fact            ; Recursive call
    MOV r12,r0          ; Restore original n
    LDR r0,[sp,#0]      ; and return address
    LDR lr,[sp,#4]      ; pop 2 items from stack
    ADD sp,sp #8        ;
    MUL r0,r0,r12       ; Multiply to get result
    MOV pc,lr           ; and return
```

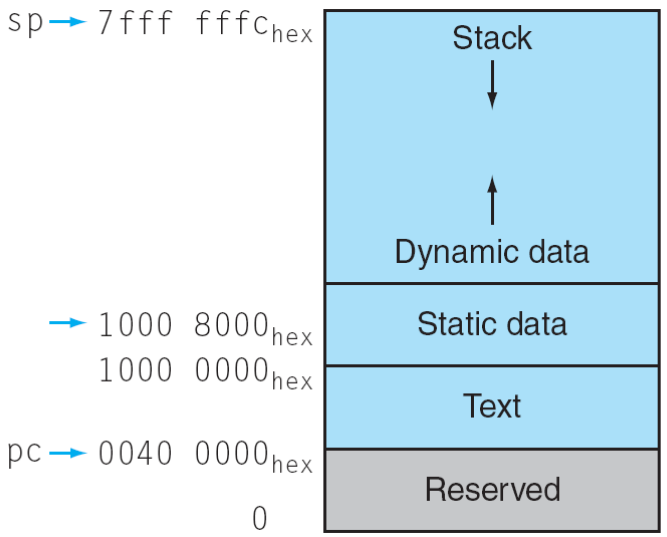
Local Data on the StackL



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory LayoutL

- 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



Character DataL

- 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

- Could use bitwise operations
- ARM byte load/store
  - String processing is a common case

```
LDRB r0, [sp,#0]  ; Read byte from source
STRB r0, [r10,#0] ; Write byte to destination
```

  - Sign extend to 32 bits

```
LDRSB ; Sign extends to fill leftmost 24 bits
```

- ARM halfword load/store

```
LDRH r0, [sp,#0]  ; Read halfword (16 bits) from source
STRH r0,[r12,#0] ; Write halfword (16 bits) to destination
```

  Sign extend to 32 bits

```
LDRSH ; Sign extends to fill leftmost 16 bits
```

String Copy ExampleL

- C code (naïve):
  - Null-terminated string

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

  - *Addresses of x, y in registers r0, r1*
  - *i in register r4*

String Copy ExampleL

- ARM code:

```
strcpy:
    SUB sp,sp, #4          ; adjust stack for 1 item
    STR r4,[sp,#0]         ; save r4
    MOV r4,#0              ; i = 0 + 0
L1: ADD r2,r4,r1           ; addr of y[i] in r2
    LDRB  r3, [r2, #0]     ; r3 = y[i]
    ADD r12,r4,r0 ;        ; Addr of x[i] in r12
    STRB r3 [r12, #0]      ; x[i] = y[i]
    BEQ L2                 ; exit loop if y[i] == 0
    ADD r4,r4,#1           ; i = i + 1
    B L1                   ; next iteration of loop
L2: LDR r4, [sp,#0]        ; restore saved r4
    ADD sp,sp, #4          ; pop 1 item from stack
    MOV pc,lr              ; return
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
- Load 32 bit constant to r4

0000 0000 1101 1001 0000 0000 0000 0000

Cond	F	I	Opcode	S	Rn	Rd	rotate-imm	mm_8
14	0	1	13	0	0	4	8	217
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	4 bits	8 bits

The 8 non-zerobits ($1101\ 1101_2$, 217_{ten}) of the constant is rotated by 16 bits and MOV instruction (opcode -13) loads the 32 bit value

Branch Instruction format

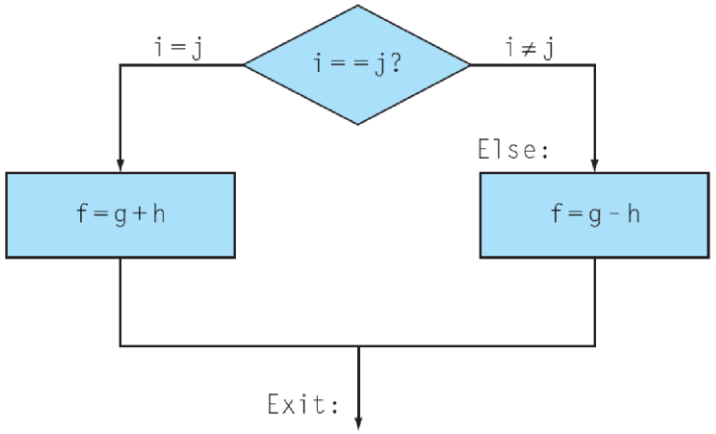
- Encoding of options for Condition field

Value	Meaning	Value	Meaning
0	EQ (EQual)	8	HI (unsigned HIgher)
1	NE (Not Equal)	9	LS (unsigned Lower or Same)
2	HS (unsigned Higher or Same)	10	GE (signed Greater than or Equal)
3	LO (unsigned LOwer)	11	LT (signed Less Than)
4	MI (MInus, <0)	12	GT (signed Greater Than)
5	PL - (PLus, >=0)	13	LE (signed Less Than or Equal)
6	VS (oVerflow Set, overflow)	14	AL (Always)
7	VC (oVerflow Clear, no overflow)	15	NV (reserved)

Conditional ExecutionL

- ARM code for executing ***if*** statement
- Code on right does not use branch. This can help performance of pipelined computers (Chapter 4)

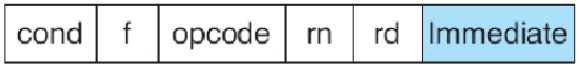
```
        CMP r3, r4              ; reg  r3 and r4
contain i,j
        BNE Else                ; go to Else if i
<> j
        ADD r0,r1,r2        ; f = g + h
        B Exit          ; go to Exit
Else: SUB r0,r1,r2          ; f = g – h
Exit:
```



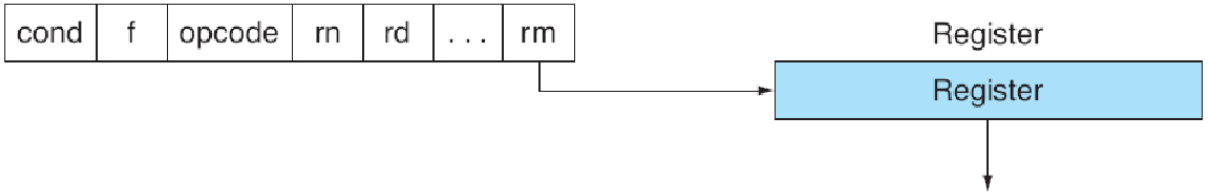

```
CMP r3,r4
ADDEQ r0,r1,r2 ; f =
g+h
SUBNE r0,r1,r2 ; f = g-
h
```

Addressing Mode Summary (1-3 of 12)

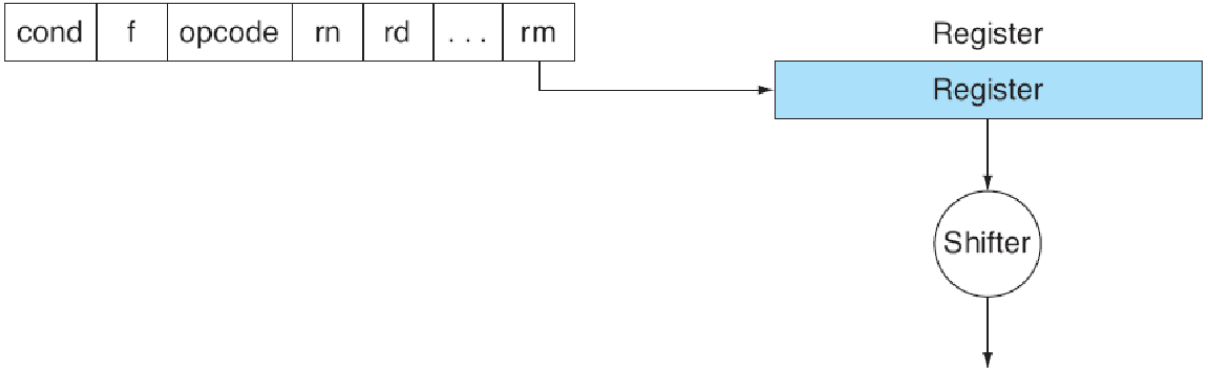
1. Immediate: ADD r2, r0, #5



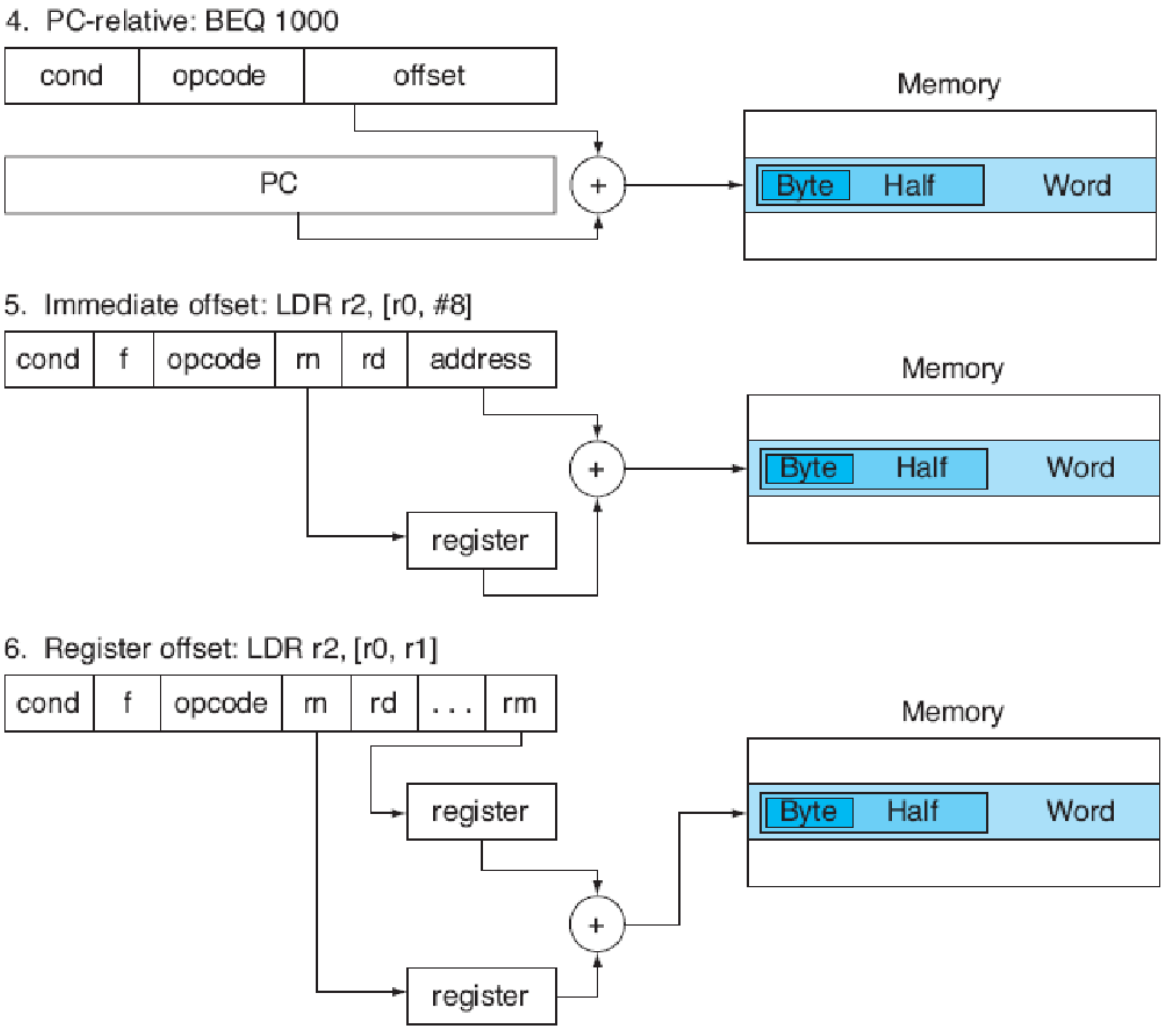
2. Register: ADD r2, r0, r1



3. Scaled register: ADD r2, r0, r1, LSL #2

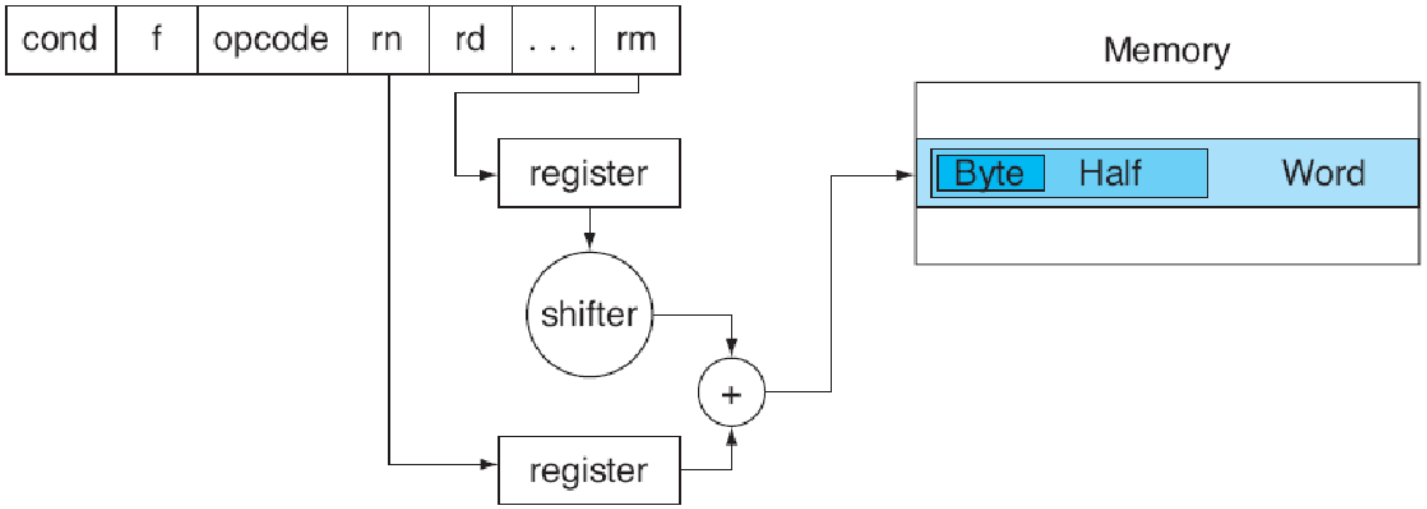


Addressing Mode Summary (4-6 of 12)

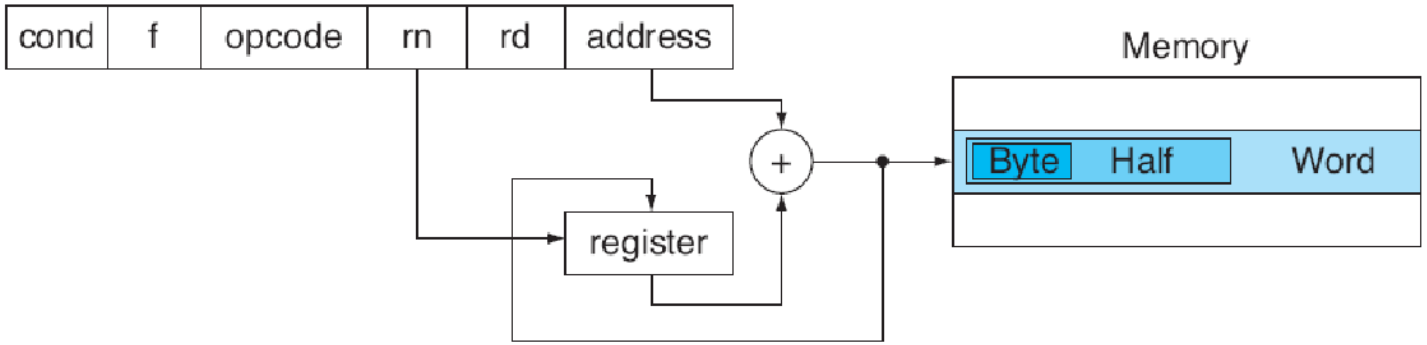


Addressing Mode Summary (7-8 of 12)

7. Scaled register offset: LDR r2, [r0, r1, LSL #2]

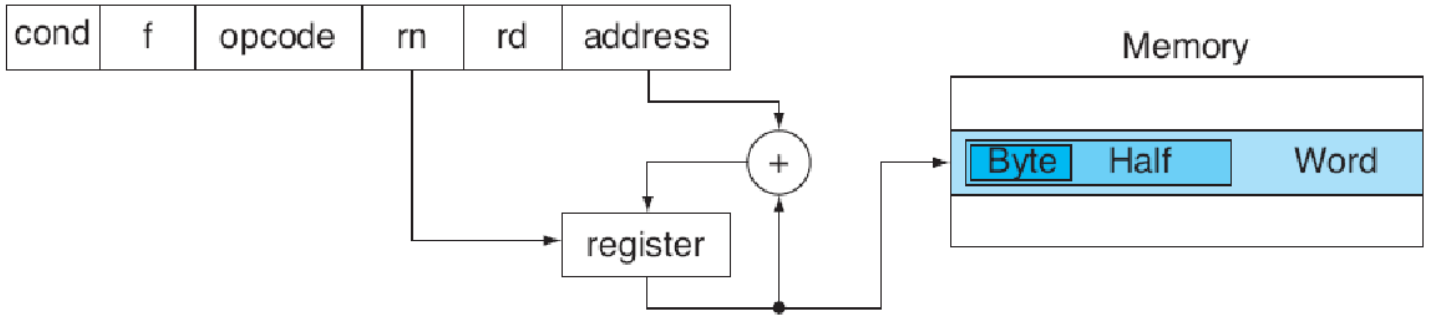


8. Immediate offset pre-indexed: LDR r2, [r0, #4]!

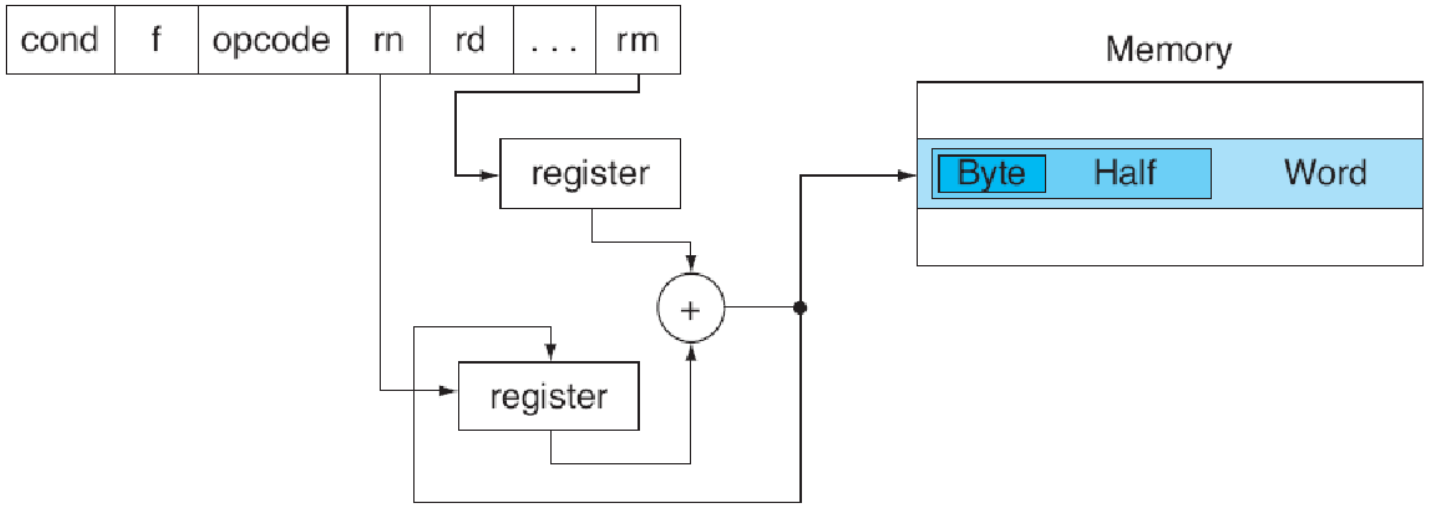


Addressing Mode Summary (9-10 of 12)

9. Immediate offset post-indexed: LDR r2, [r0], #4

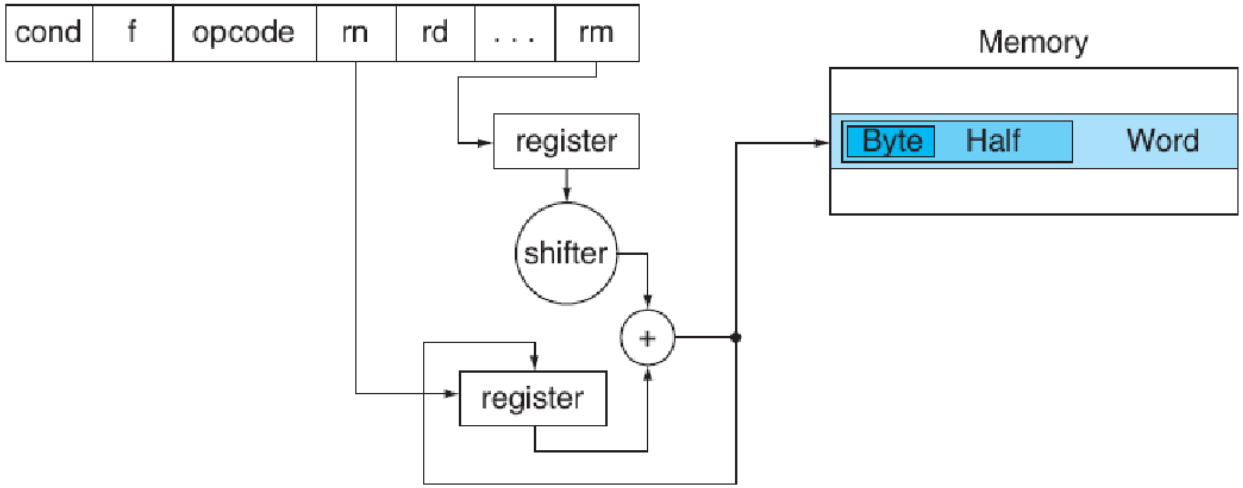


10. Register offset pre-indexed: LDR r2, [r0, r1]!

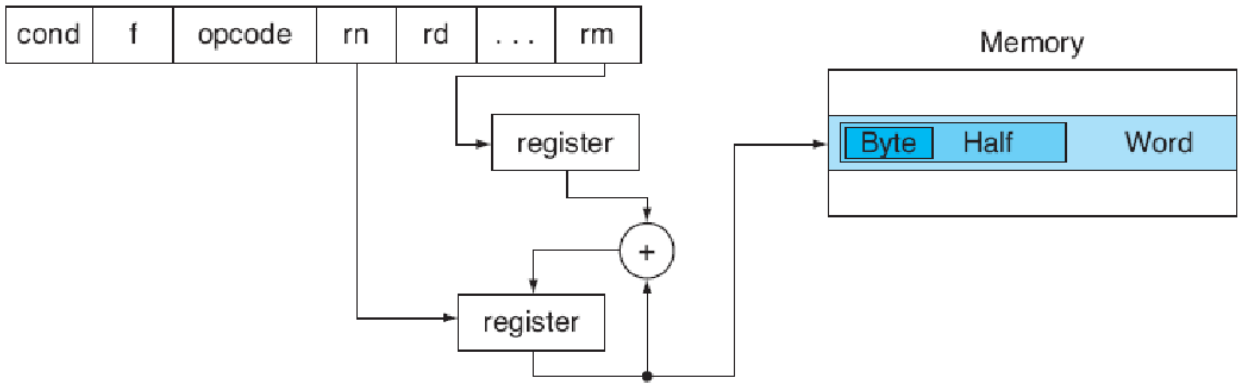


Addressing Mode Summary (11-12)

11. Scaled register offset pre-indexed: LDR r2, [r0, r1, LSL #2]!



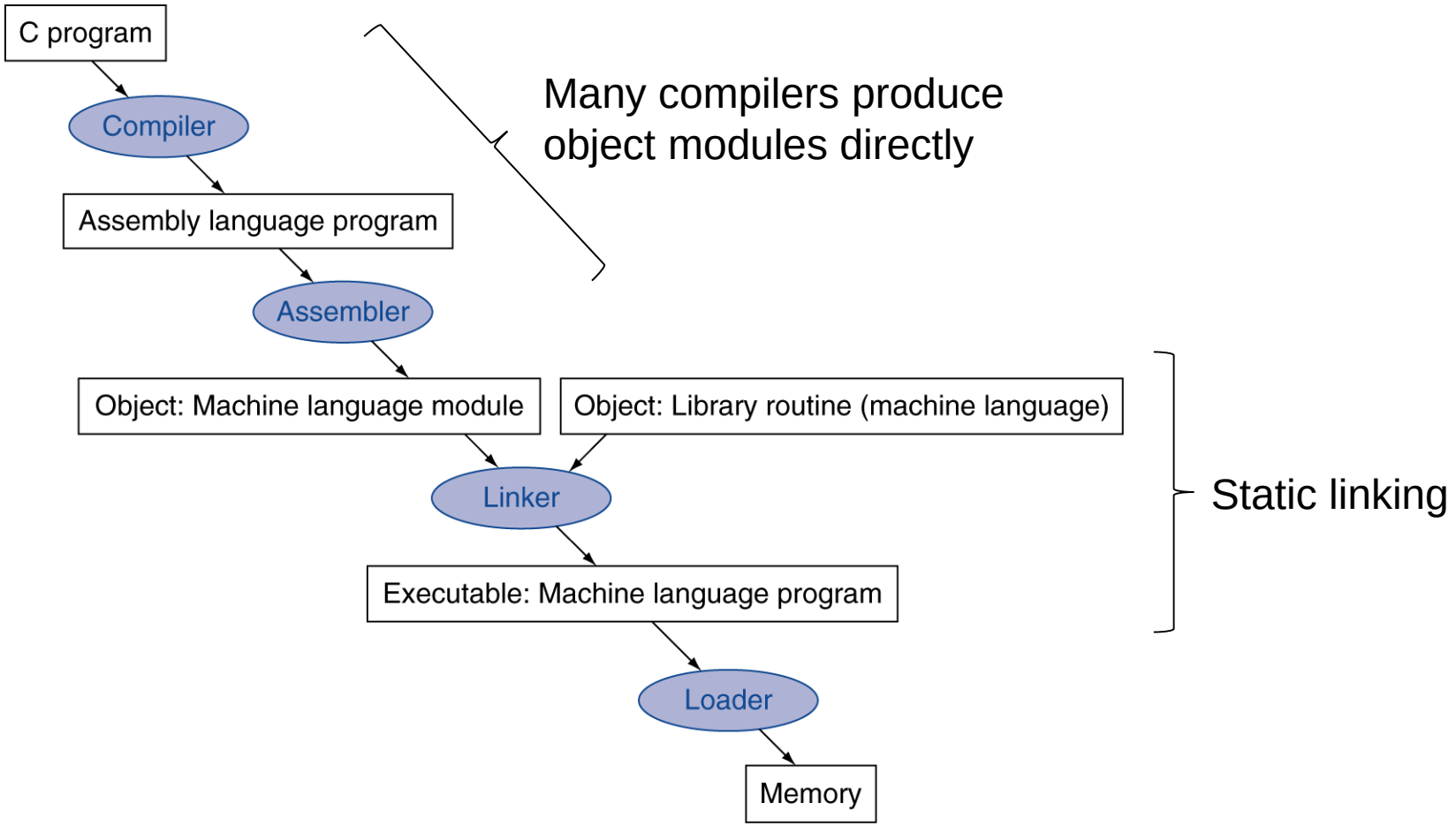
12. Register offset post-indexed: LDR r2, [r0], r1



SynchronizationX

- 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
- Single instruction for atomic exchange or swap
  - Atomic swap of register ↔ memory
  - ARM instruction: SWP

Translation and StartupL



Assembler PseudoinstructionsL

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

  LDR r0, #constant

The assembler determines which instructions to use to create the constant in the most efficient way.

Producing an Object ModuleL

- 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 ModulesL

- 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 ProgramL

- 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
  6. Jump to startup routine
     - Copies arguments to r0, … and calls main
     - When main returns, startup terminates with exit system call

Dynamic LinkingL

- 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

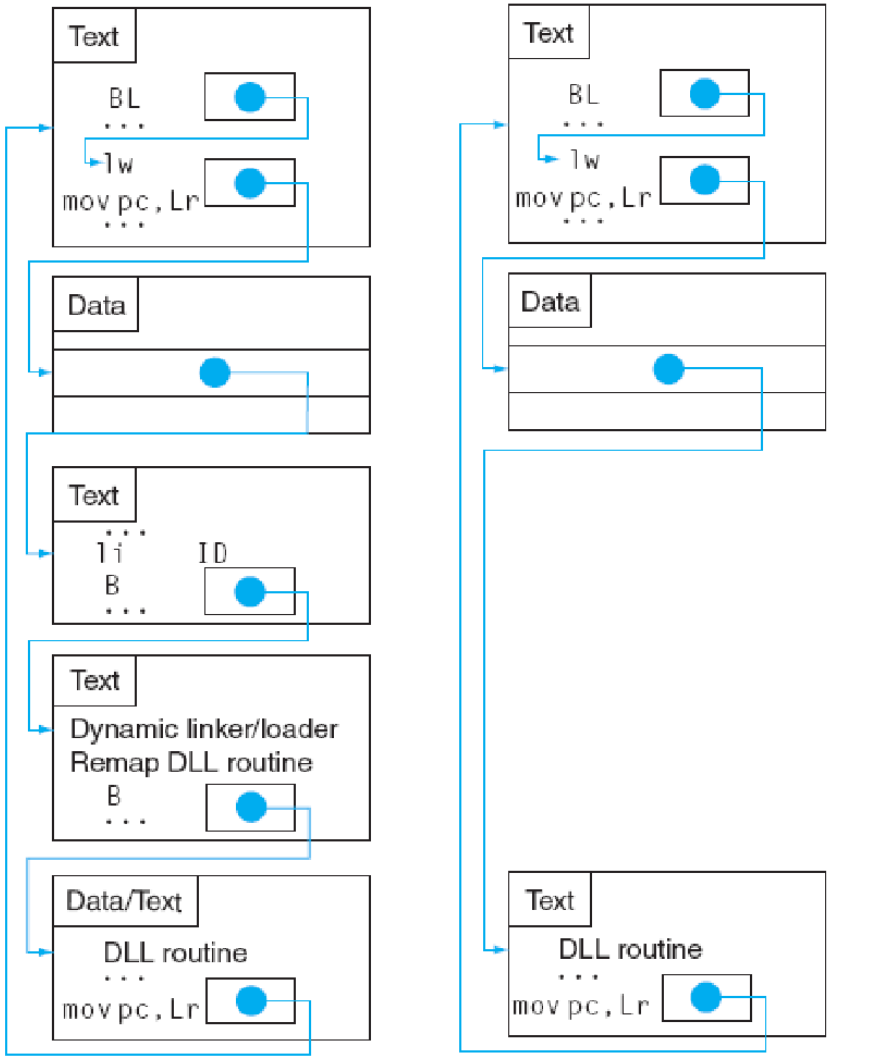
Lazy LinkageL

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

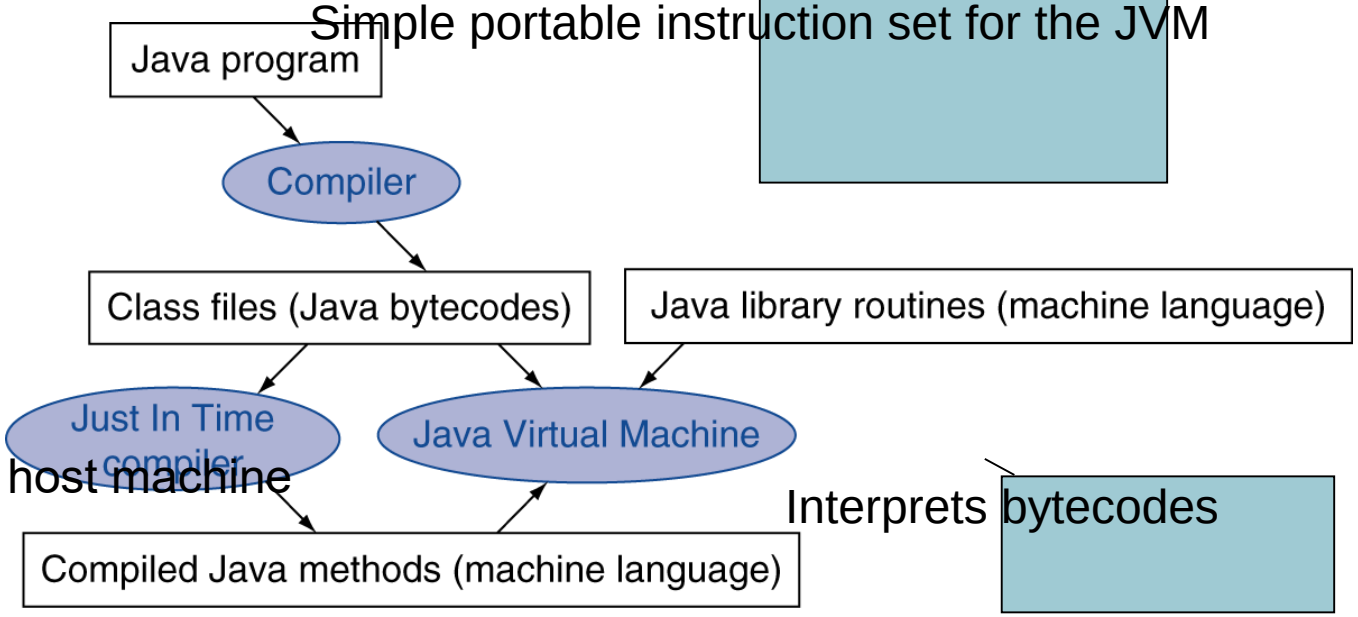
Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

Starting Java ApplicationsX



C Sort ExampleL

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

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

The Procedure SwapL

Assembler directive

```
v       RN 0     ; 1st argument address of v
k       RN 1     ; 2nd argument index k
temp    RN 2     ; local variable
temp2   RN 3     ; temporary variable for v[k+1]
vkAddr  RN 12    ; to hold address of v[k]
```

Procedure body

```
swap: ADD    vkAddr, v, k, LSL #2  ; reg vkAddr = v + (k * 4)
                                   ; reg vkAddr has the address of v[k]
      LDR    temp, [vkAddr, #0]    ; temp (temp) = v[k]
      LDR    temp2, [vkAddr, #4]   ; temp2 = v[k + 1]
                                   ; refers to next element of v
      STR    temp2, [vkAddr, #0]   ; v[k] = temp2
      STR    temp, [vkAddr, #4]    ; v[k+1] = temp
```

Procedure return

```
      MOV    pc, lr                ; return to calling routine
```

The Sort Procedure in CL

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int 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);
    }
  }
}
```

Register allocation and saving registers for *sortL*

Register allocation

v	RN 0	; 1st argument address of v
n	RN 1	; 2nd argument index n
i	RN 2	; local variable i
j	RN 3	; local variable j
vjAddr	RN 12	; to hold address of v[j]
vj	RN 4	; to hold a copy of v[j]
vj1	RN 5	; to hold a copy of v[j+1]
vcopy	RN 6	; to hold a copy of v
ncopy	RN 7	; to hold a copy of n

Saving registers

```
sort:   SUB   sp,sp,#20          ; make room on stack for 5 registers
        STR   lr, [sp, #16]      ; save lr on stack
        STR   ncopy, [sp, #12]   ; save ncopy on stack
        STR   vcopy,  [sp, #8]   ; save vcopy on stack
        STR   j,  [sp, #4]       ; save j on stack
        STR   i,  [sp, #0]       ; save i on stack
```

Procedure body - *sortL*

Move parameters		MOV	vcopy, v	; copy parameter v into vcopy (save r0)
		MOV	ncopy, n	; copy parameter n into ncopy (save r1)
Outer loop		MOV	i, #0	; i = 0
	for1tst:	CMP	i, n	; if i ≥ n
		BGE	exit1	; go to exit1 if i ≥ n
Inner loop		SUB	j, i, #1	; j = i - 1
	for2tst:	CMP	j, #0	; if j < 0
		BLT	exit2 ; go to exit2 if j < 0	
		ADD	vjAddr, v, j, LSL #2	; reg vjAddr = v + (j * 4)
		LDR	vj, [vjAddr,#0]	; reg vj = v[j]
		LDR	vj1, [vjAddr,#4]	; reg vj1 = v[j + 1]
		CMP	vj, vj1	; if vj ≤ vj1
		BLE	exit2	; go to exit2 if vj ≤ vj1
Pass parameters and call		MOV	r0, vcopy	; first swap parameter is v
		MOV	r1, j	; second swap parameter is j
		BL	swap	; swap code shown in Figure 2.23
Inner loop		SUB	j, j, #1	; j -= 1
		B	for2tst	; branch to test of inner loop
Outer loop	exit2:	ADD	i, i, #1	; i += 1
		B	for1tst	; branch to test of outer loop

Restoring registers and return - *sortL*

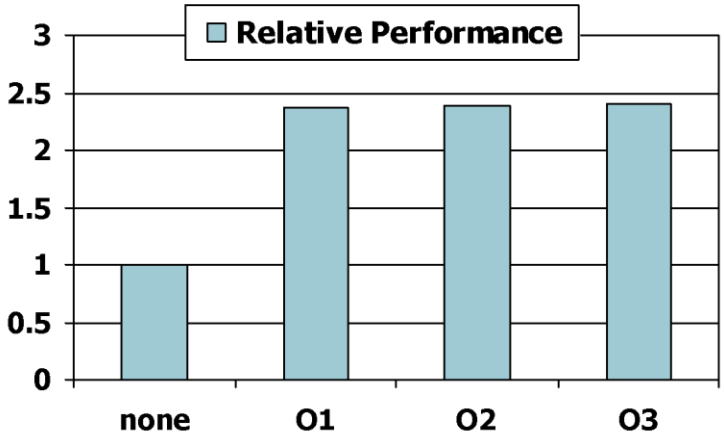
```
exit1:   LDR    i,  [sp, #0]          ; restore i from stack
         LDR    j,  [sp, #4]          ; restore j from stack
         LDR    vcopy,  [sp, #8]      ; restore vcopy from stack
         LDR    ncopy, [sp, #12]      ; restore ncopy from stack
         LDR    lr, [sp, #16]         ; restore lr from stack
         ADD    sp,sp,#20             ; restore stack pointer
```

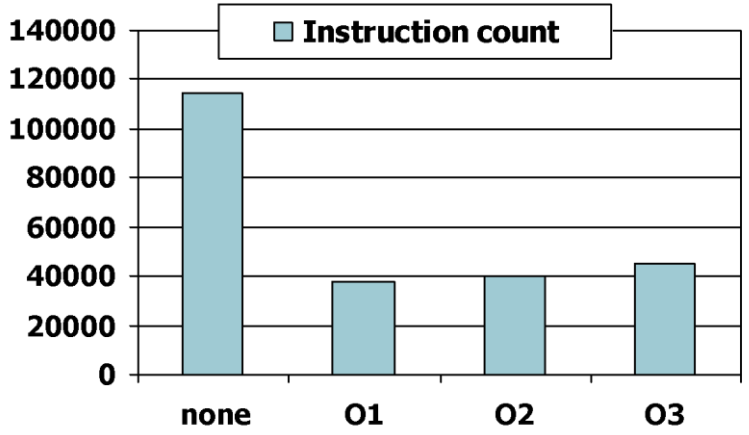
Procedure return

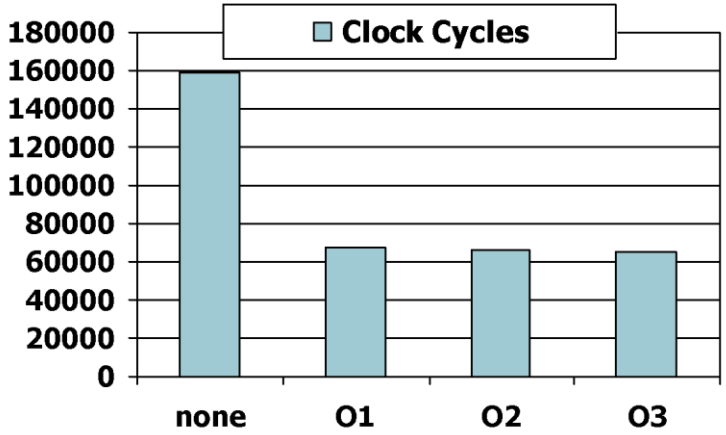
```
         MOV    pc, lr                ; return to calling routine
```

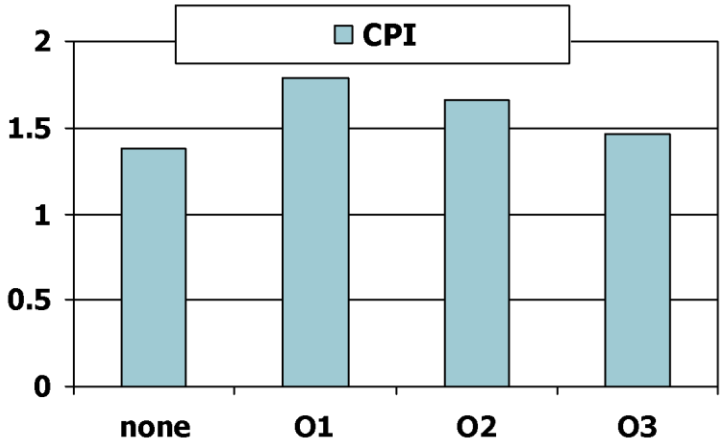
Effect of Compiler OptimizationL

Compiled with gcc for Pentium 4 under Linux

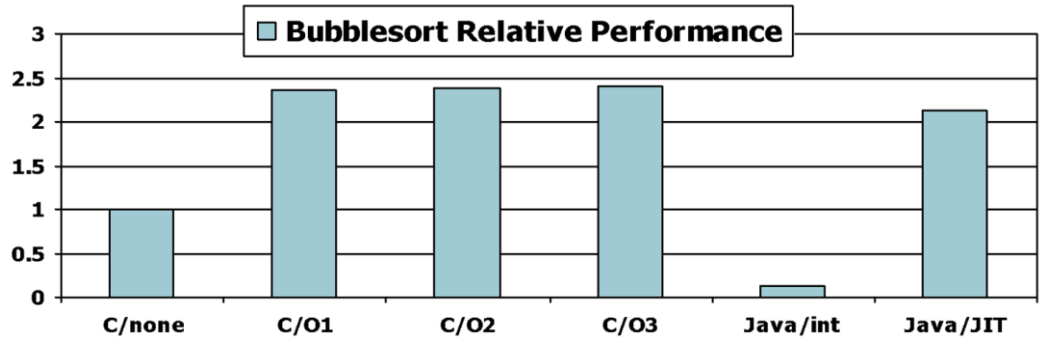


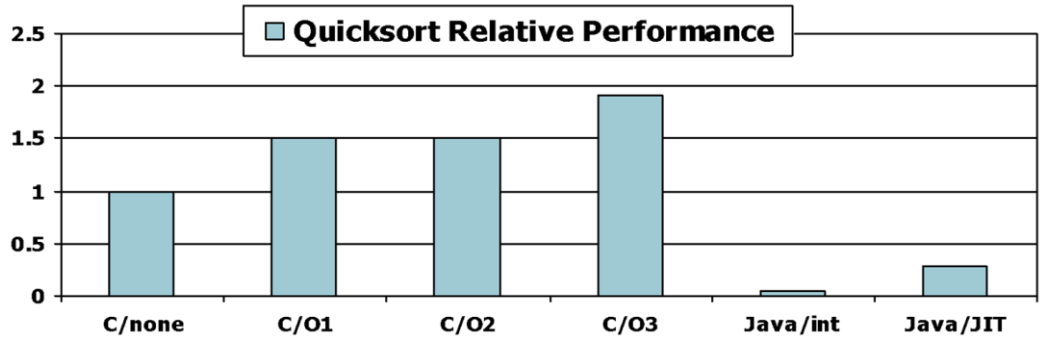


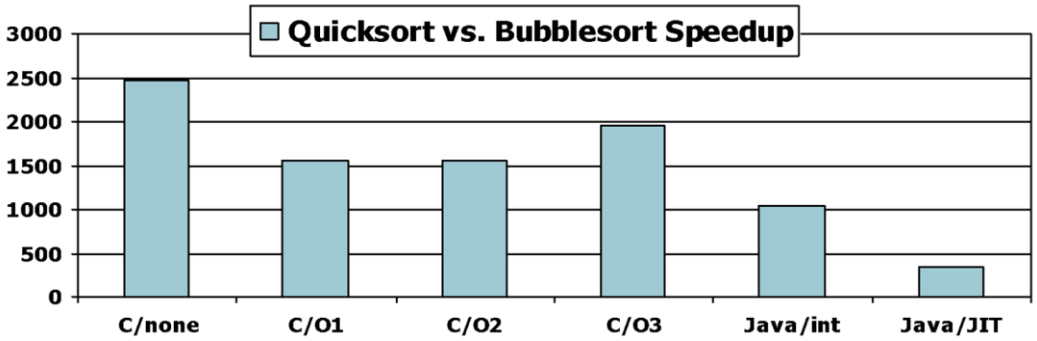




Effect of Language and AlgorithmL







Lessons LearntL

- 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. PointersL

- 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 and ArrayL

```
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   i,#0      ; i = 0
      MOV   zero,#0   ;zero = 0
loop1: STR  zero, [array,i, LSL #2] ; array[i] = 0
      ADD   i,i,#1    ; i = i + 1
      CMP   i,size    ; i < size
      BLT   loop1     ; if (i < size) go to loop1
```

```
      MOV   p,array     ; p = & array[0]
      MOV   zero,#0     ;zero = 0
      ADD   arraySize,array,size,LSL #2
                        ; arraySize = &array[size]
loop2: STR  zero,[p],#4  ; Memory[p] = 0,p = p + 4
      CMP   p,arraySize; p<&array[size]
      BLT  loop2        ; if () go to loop2
```

Comparison of Array vs. PtrL

- 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

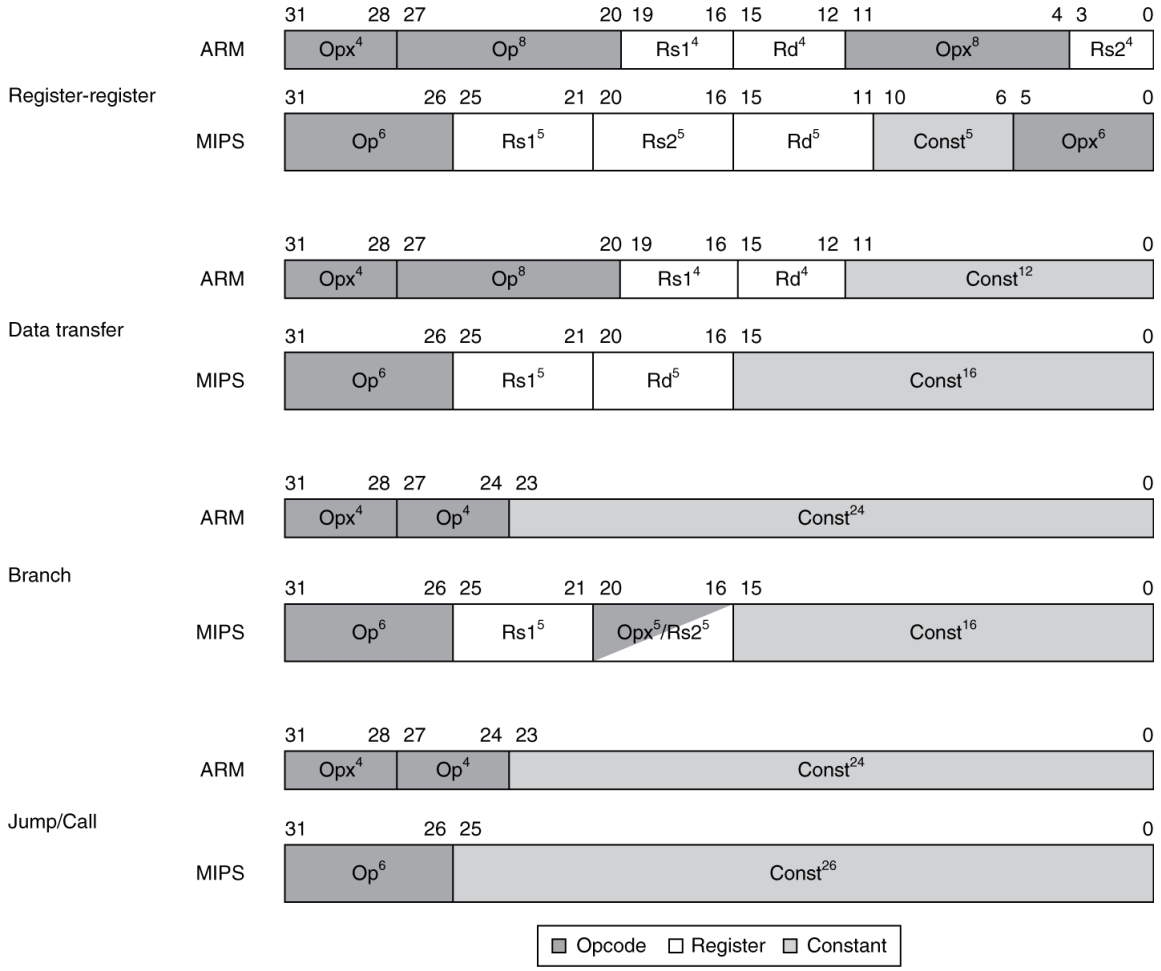
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	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

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Instruction Encoding



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

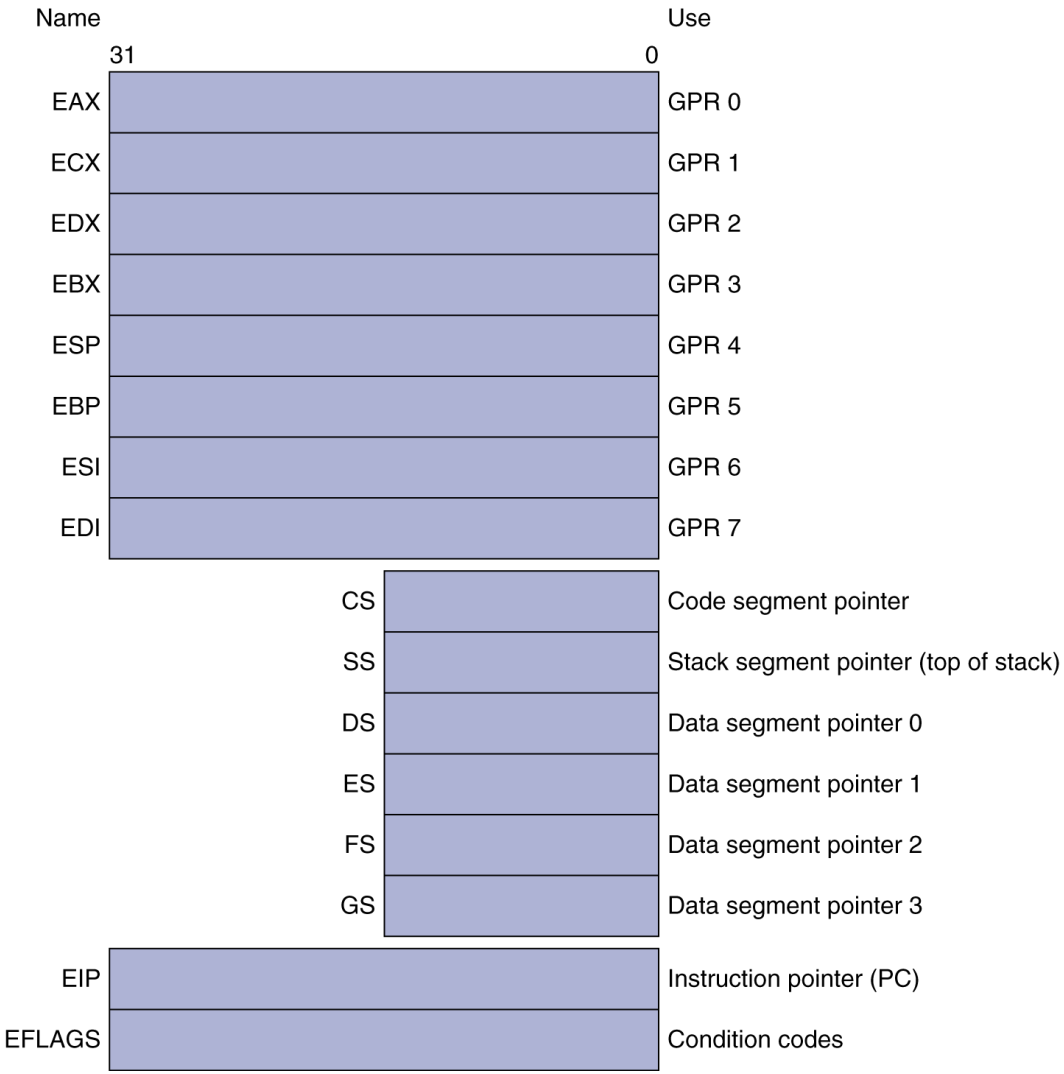
The Intel x86 ISA

- 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



Basic x86 Addressing Modes

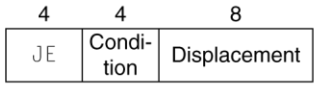
- Two operands per instruction

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

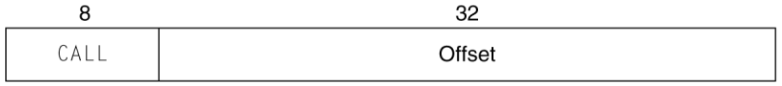
- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding

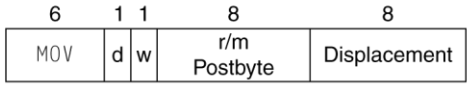
a. JE EIP + displacement



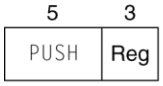
b. CALL



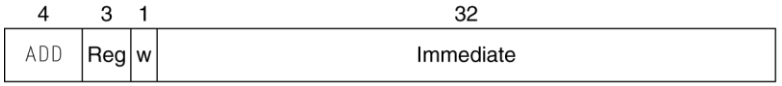
c. MOV EBX, [EDI + 45]



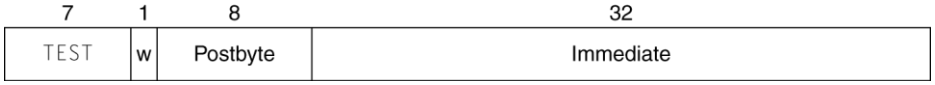
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



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

Implementing IA-32

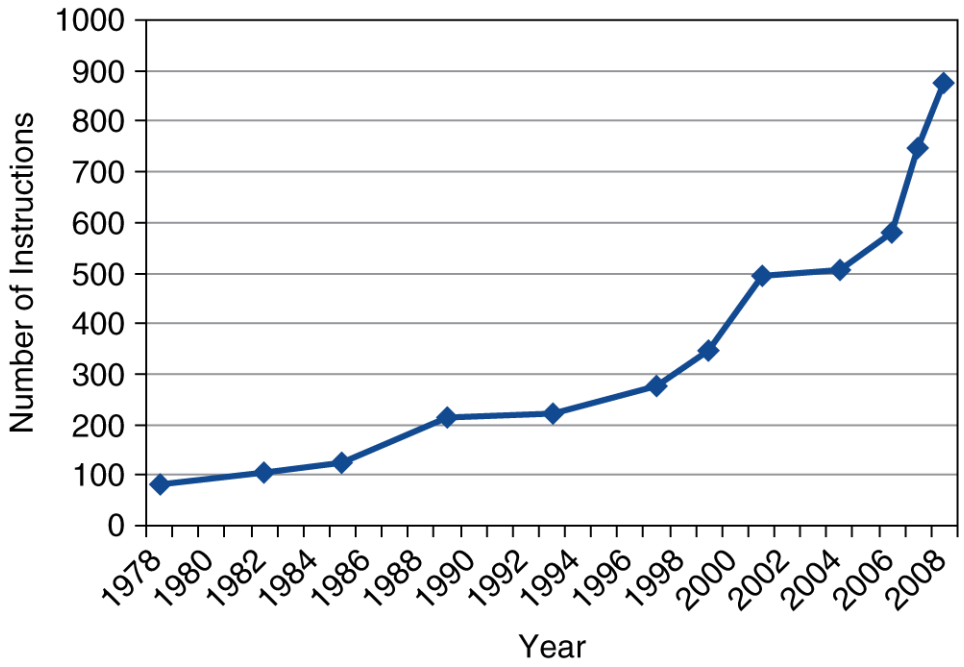
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - 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 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 –MIPS ISA
  2. Smaller is faster – MIPS GPR , 32
  3. Make the common case fast – PC-re-addr, immediate constant
  4. Good design demands good compromises – providing imme-addr, const, & same inst size
- Layers of software/hardware
  - Compiler, assembler, hardware
- ARM: typical of RISC ISAs
  - c.f. x86

Concluding Remarks

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

Instruction class	ARM examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	ADD,SUB,MOV	16%	48%
Data transfer	LDR,STR,LDRB,LDRSB,LDRH, LDRSH,STRB,STRH	35%	36%
Logical	AND,ORR,MNV,LSL,LSR	12%	4%
Conditional Branch	B_,CMP	34%	8%
Jump	B,BL	2%	0%