Lecture 9

CMPEN 331

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC (program counter)-relative addressing
  - Target address = PC + branch address

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

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

Address						
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
      beq $s0,$s1, L1
              ↓
      bne $s0,$s1, L2
      j L1
L2:   …
```

Addressing Modes

- Addressing modes are the ways of specifying an operand or a memory address.
- It is how an address (memory or register) is determined.
- Instruction type is how the instruction is put together.
- Example: addi, beq, and lw are all I-types instructions.
  - addi uses immediate addressing mode
  - beq uses pc-relative addressing
  - lw uses base addressing

Addressing Modes

- **IMMEDIATE:** a numeric value embedded in the instruction is the actual operand.
- **REGISTER:** a source or destination operand is specified as content of one of the registers *$0-$31.* This is used in the **jr** (jump register) instruction
- **PC-RELATIVE:** a data or instruction memory location is specified as an offset relative to the incremented PC. This is used in the **beq** and **bne** (branch equal, branch not equal) instructions.
- **BASE:** a data or instruction memory location is specified as assigned offset from a register. This is used in the **lw** and **sw** (load word, store word) instructions.
- **PSEUDO-DIRECT:** the memory address is(mostly) embedded in the instruction. This is used in the **j** (jump) instruction.

Addressing Modes

- Register addressing

  Operand is in register

  add \$s1, \$s2, \$s3 means \$s1 ← \$s2 + \$s3

- Base addressing

  Operand is in memory.

  The address is the sum of a register and a constant.

  lw \$s1, 32(\$s3) means \$s1 ← M[s3 + 32]

- Immediate addressing

  The operand is a constant.

  addi \$s1, \$zero, 7 means \$s1 ← 0 + 7

- PC-relative addressing

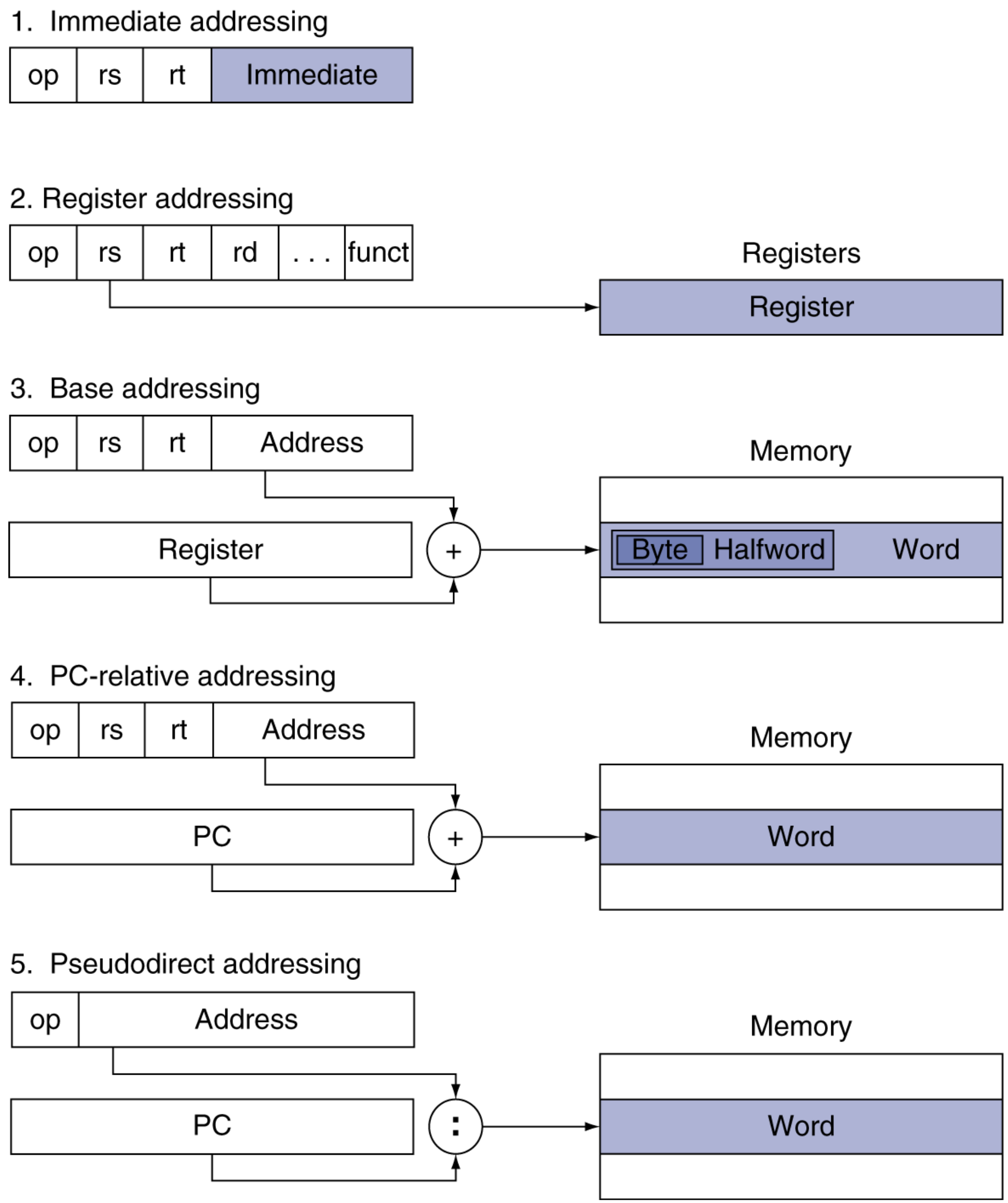
  The operand address = PC + an offset

  Implements position-independent codes.

- Pseudo-direct addressing

  Used in the J format. The target address is the concatenation of the 4 MSB's of the PC with the 28-bit offset. This is a minor variation of the PC-relative addressing format.

Addressing Mode Summary



C Swap Example

- 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;
}
```

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions:

```
move $t0, $t1      → add $t0, $zero, $t1
blt $t0, $t1, L    → slt $at, $t0, $t1
                     bne $at, $zero, L
```

  - \$at (register 1): assembler temporary

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 and Array

<table>
<tr>
<td><pre>clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}</pre></td>
<td><pre>clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}</pre></td>
</tr>
<tr>
<td><pre>   move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2   # $t1 = i * 4
       add $t2,$a0,$t1 # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1 # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1</pre></td>
<td><pre>   move $t0,$a0   # p = & array[0]
       sll $t1,$a1,2   # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2</pre></td>
</tr>
</table>

What is a pointer

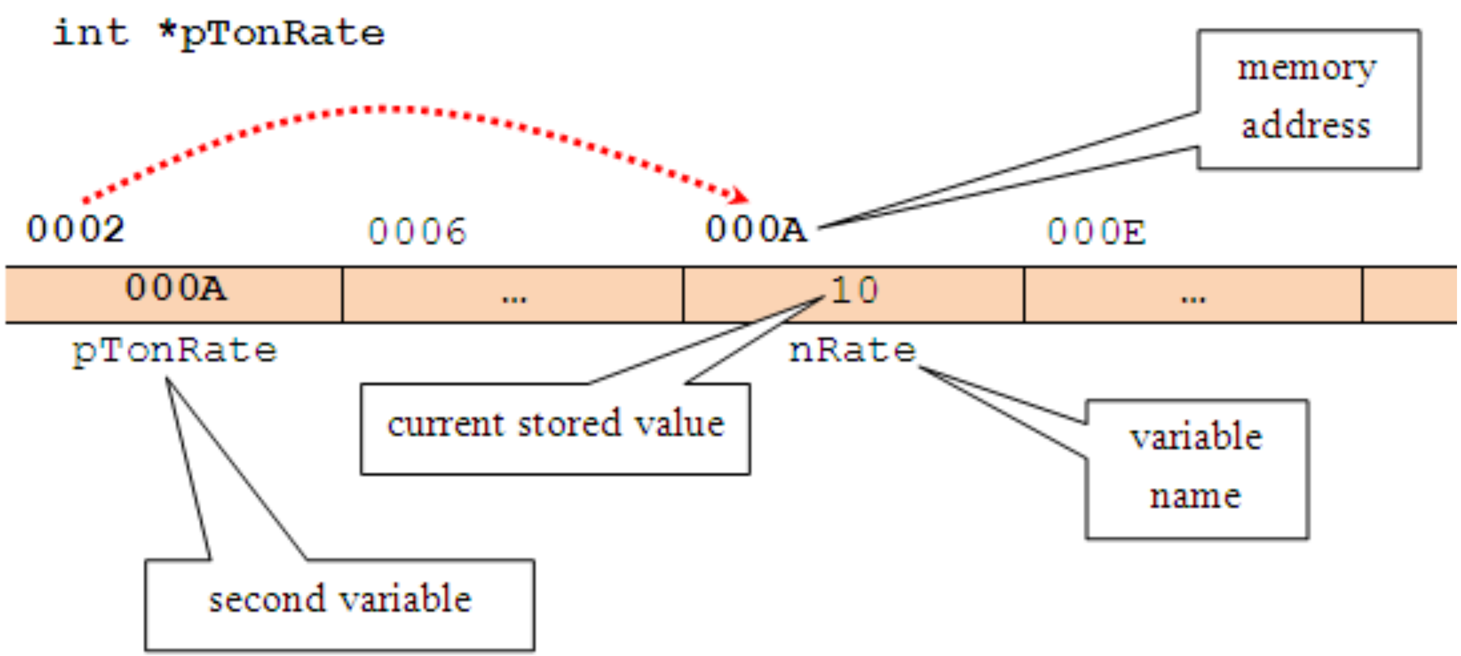
- In a generic sense, a “pointer” is anything that tells us where something can be found.
- When declaring a variable, the compiler sets aside memory storage with a unique address to store that variable.
- The compiler associates that address with the variable’s name.
- We can manipulate the memory address by using pointers which means that we create a second variable for storing the memory address.

Pointers

- Let us store the `nRate`'s memory address, in `pTonRate` variable.
- So, `pTonRate` now holds the `nRate`'s memory address, where the actual data (10) is stored.
- Pointer variable declaration becomes something like this,

```
int    *pTonRate;
```

- The asterisk (`*`) is used to show that is it the pointer variable instead of normal variable.



www.tenouk.com, ©

Pointers

- A variable name *directly* references a value.
- A pointer *indirectly* references a value. Referencing a value through a pointer is called *indirection*.
- A pointer variable must be declared before it can be used.
- C uses two pointer operators,
  1. Indirection operator (`*`) – asterisk symbol, has been explained previously.
  2. Address-of-operator (`&`) – ampersand symbol, means return the address of…
- When `&` operator is placed before the name of a variable, it will returns the memory address of the variable instead of stored value.

www.tenouk.com, ©

Pointers

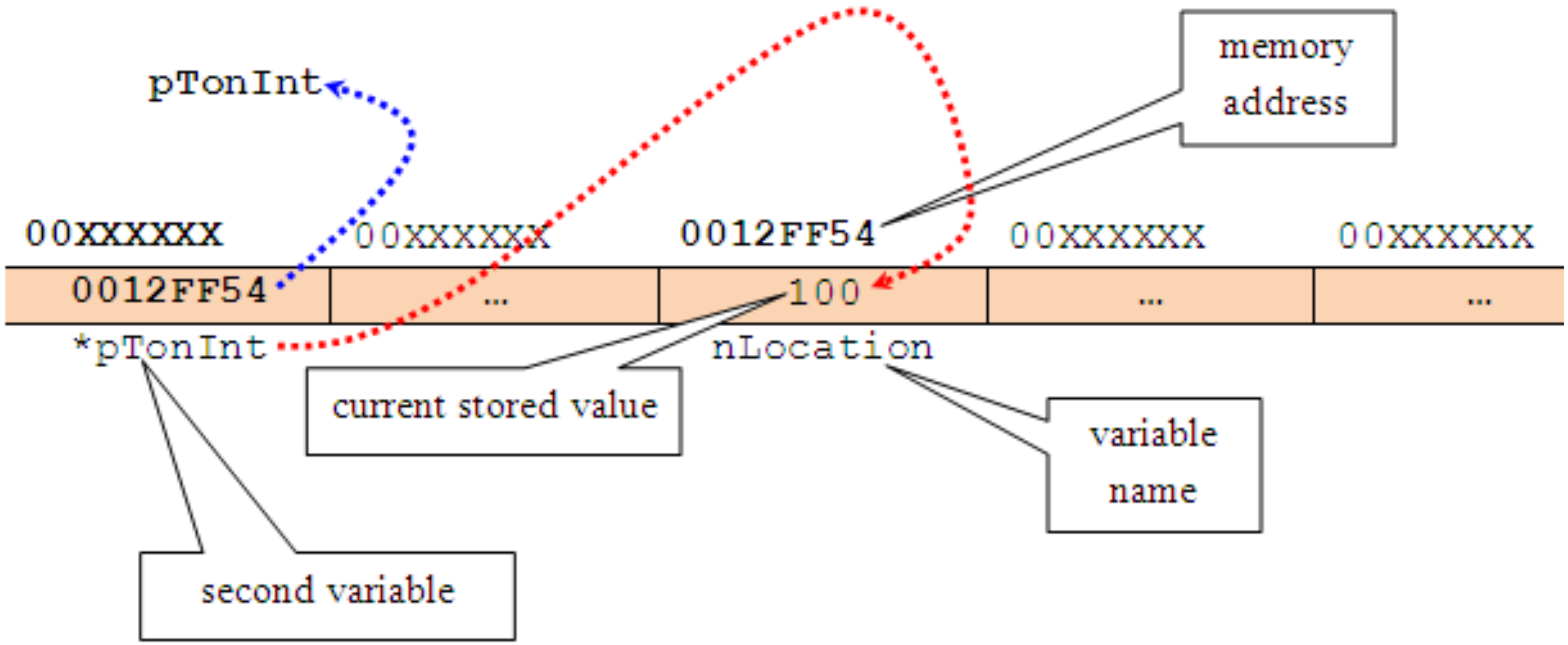
```
int x = 1, y = 2, z[10];
int *ip;            /* ip is a pointer to an int */

ip = &x;            /* ip points to (contains the memory
                    address of) x */
```

- & "address operator" which gives or produces the memory address of a data variable
- * "dereferencing operator" which provides the contents in the memory location specified by a pointer
- The * operator is a complement of & operator.

www.tenouk.com, ©

Pointers

```
int  * pToInt;
pToInt = &nLocation;
nLocation = 100;
```



www.tenouk.com, ©

Example: Clearing and Array

<table>
<tr>
<td>

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

</td>
<td>

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

</td>
</tr>
<tr>
<td>

```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

</td>
<td>

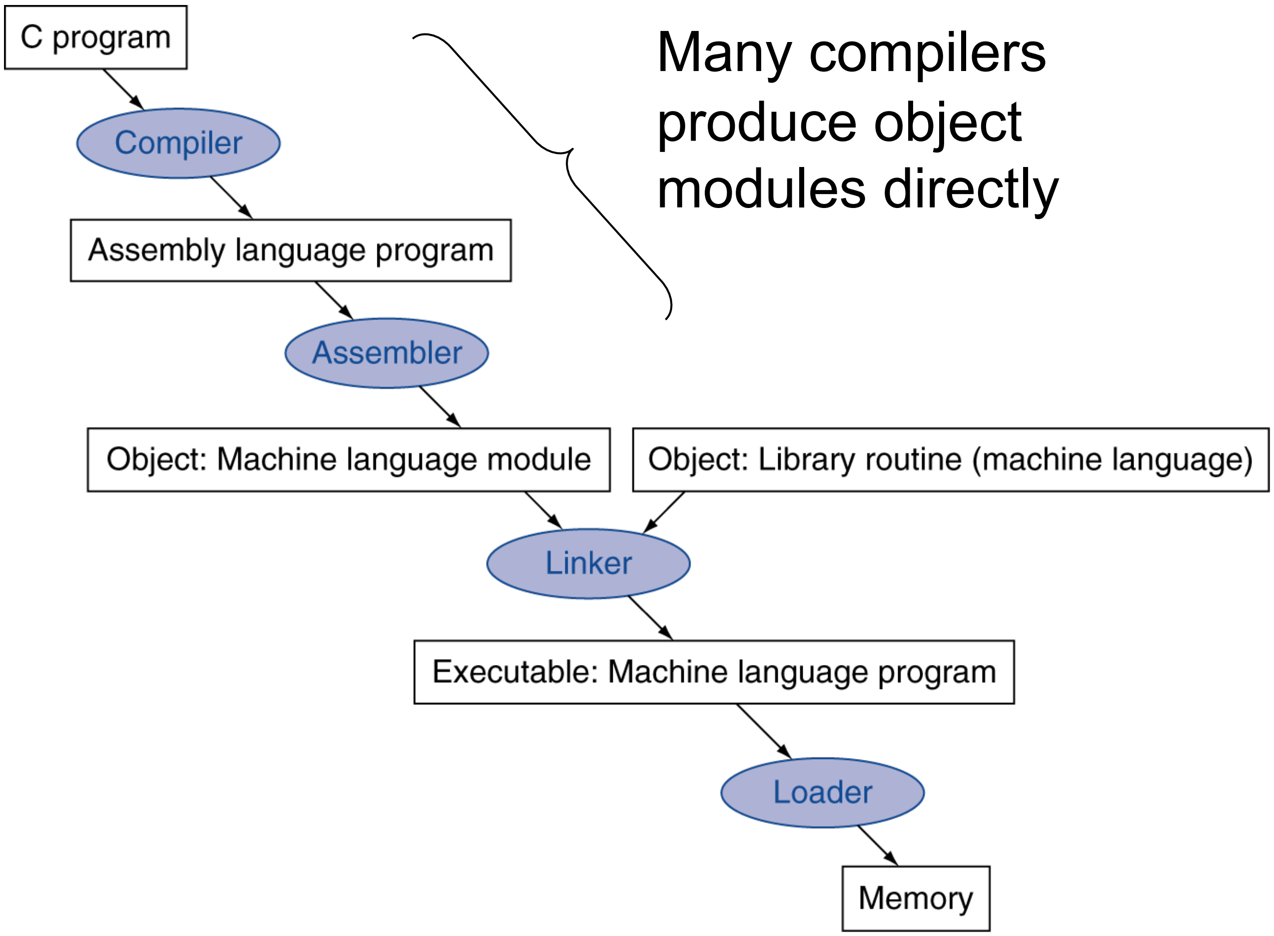
```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0)  # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2  # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

</td>
</tr>
</table>

Lecture 10

CMPEN 331

Translation and Startup



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
  - 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

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, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit system call

Dynamic Linking

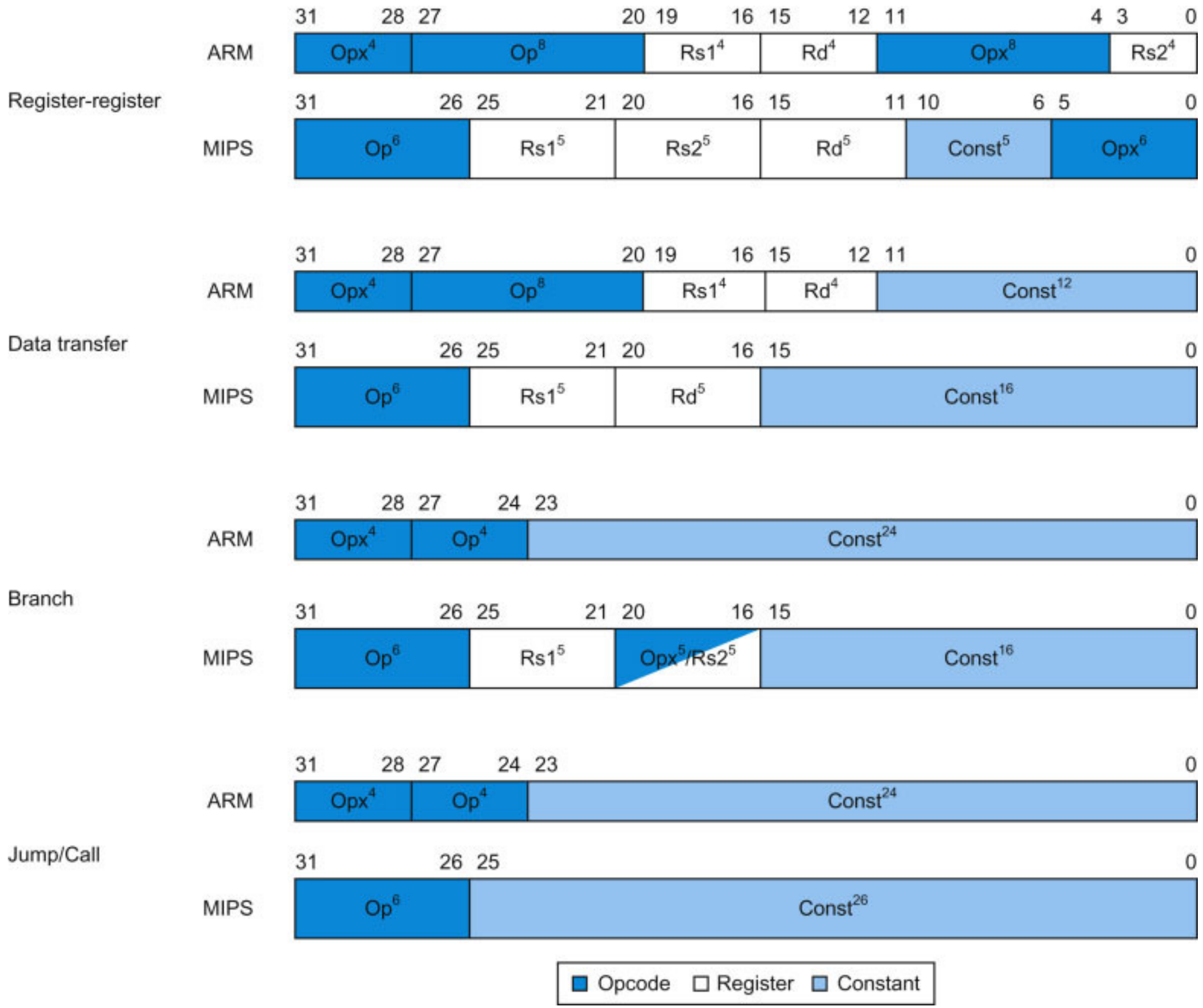
- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

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	32 × 32-bit
Input/output	Memory mapped	Memory mapped

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
    - 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

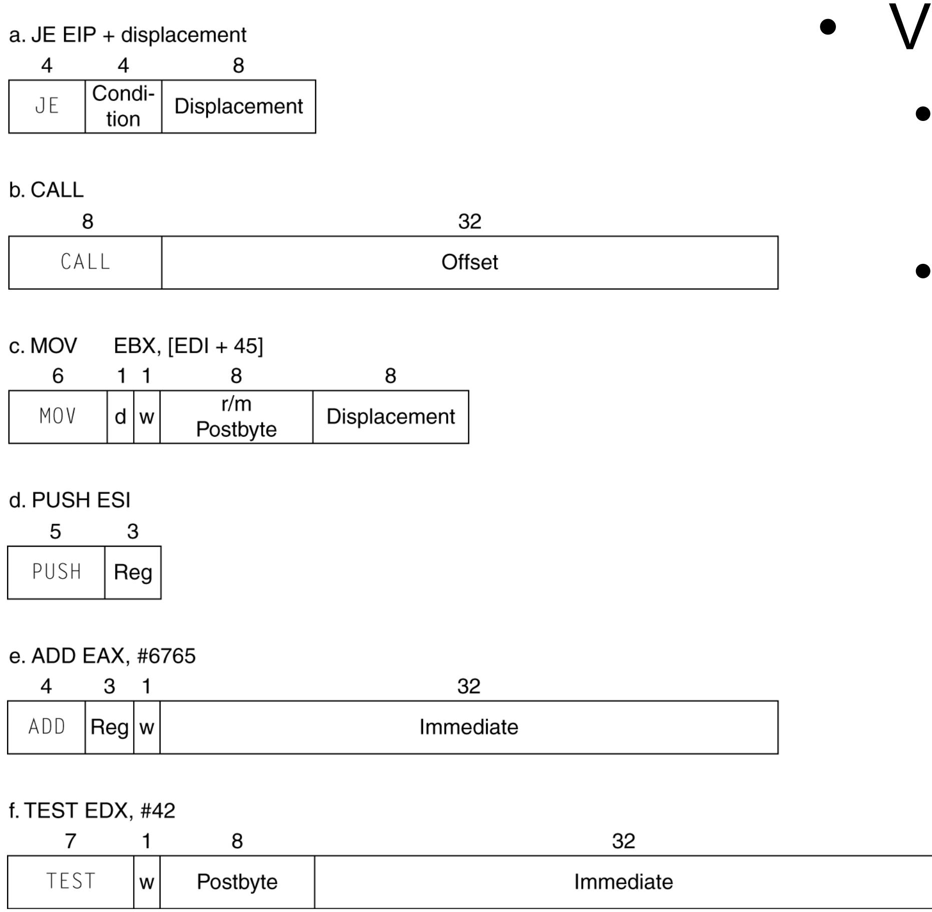
The Intel x86 ISA

- 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!

Basic x86 Registers

Name (31 … 0)	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

x86 Instruction Encoding



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

x86 Typical Operation

Instruction	Meaning
Control	**Conditional and unconditional branches**
jnz, jz	Jump if condition to EIP + 8-bit offset; JNE (forJNZ), JE (for JZ) are alternative names
jmp	Unconditional jump—8-bit or 16-bit offset
call	Subroutine call—16-bit offset; return address pushed onto stack
ret	Pops return address from stack and jumps to it
loop	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX $\neq$ 0
Data transfer	**Move data between registers or between register and memory**
move	Move between two registers or between register and memory
push, pop	Push source operand on stack; pop operand from stack top to a register
les	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
add, sub	Add source to destination; subtract source from destination; register-memory format
cmp	Compare source and destination; register-memory format
shl, shr, rcr	Shift left; shift logical right; rotate right with carry condition code as fill
cbw	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
test	Logical AND of source and destination sets condition codes
inc, dec	Increment destination, decrement destination
or, xor	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
movs	Copies from string source to destination by incrementing ESI and EDI; may be repeated
lods	Loads a byte, word, or doubleword of a string into the EAX register

Arithmetic for Computers

Chapter 3

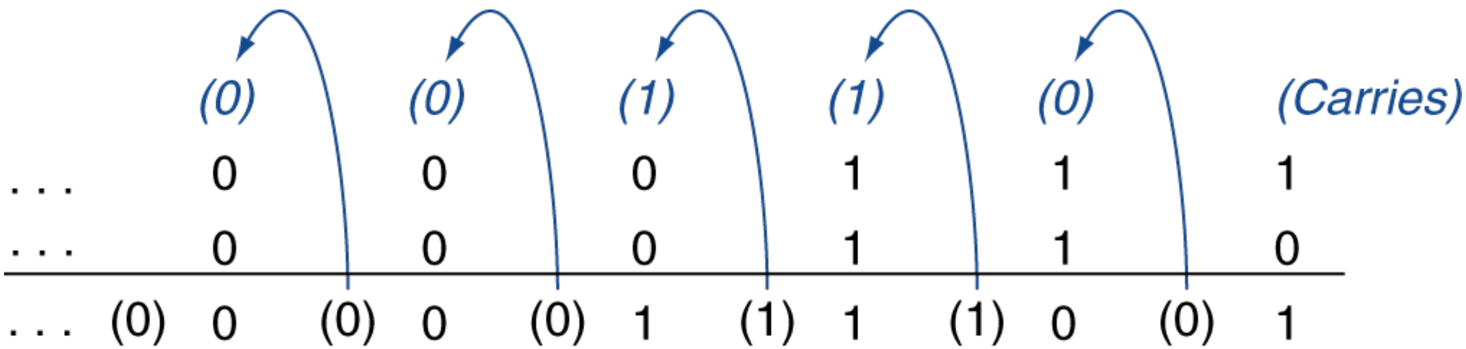
Arithmetic for Computers

§3.1 Introduction

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

§3.2 Addition and Subtraction

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addiu`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action