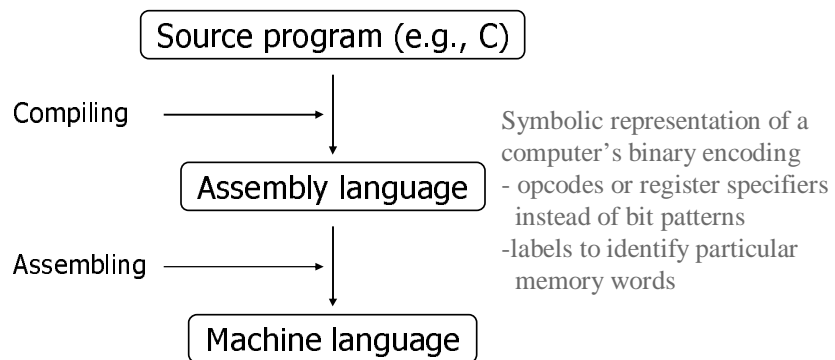

Chapter 2.10 ~ 16

2.10 Translating and Starting a Program : Compilation



Why Assembly Languages ?

- **Program speed is critically important**
 - With compilers, no control over instructions to execute
 - Thus, not sure about its response time
- **Program size is critically important**
 - Embedded applications
 - Sometimes needs to be fit into internal RAM
- **No high-level language is available**

Why Assembly Languages ?

- **GEOS from GeoWorks**
 - Object oriented operating system, entirely written in assembly language (Intel 8086)
 - **Products**
 - Geobook, PC-NewDeal, PC-Ensemble
 - HP's OmniGo
 - Sharp's Zoomer
 - Nokia's Communicator smart phones
 - **Advantages & Disadvantages ???**
- **What DSP programmers usually do ?**

How to Assemble

- “Assembling”
 - is a straightforward process compared to “compiling”
 - Machine language format corresponds to memory layout.
 - Dictates where file contents loaded into memory.
 - Segments specified in assembly.
 - `.align N`
 - Ensure following item is N-byte aligned.
 - I.e., skip bytes based on current offset.
 - `.section “...”`
 - Following info is in specified segment.
 - Need to keep track of each segment’s contents separately.
- but, “label handling” is the most difficult task

```

00100111101111011111111111110000
1010111110111111000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
1010111110100000000000000011000
1010111110100000000000000011100
1000111110101110000000000011100
1000111110111000000000000011000
0000000111001110000000000011001
0010010111001000000000000000001
0010100100000010000000001100101
1010111110101000000000000011100
000000000000000011110000010010
000000110000111110010000100001
000101000010000011111111111011
1010111110111001000000000011000
001111000000100000100000000000
1000111110100101000000000011000
000011000010000000000011101100
001001001000010000001000010000
1000111110111110000000000010100
0010011110111110000000000010000
000000111110000000000000001000
00000000000000000100000100001
  
```

```

      .text
      .align 2
      .globl main

main:
    addiu $29, $29, -32
    sw    $31, 20($29)
    sw    $4, 32($29)
    sw    $5, 36($29)
    sw    $0, 24($29)
    sw    $0, 28($29)
    lw    $14, 28($29)
    lw    $24, 24($29)
    mflr  $15
    addu  $25, $24, $15
    bne   $1, $0, -9
    sw    $25, 24($29)
    lui   $4, 4096
    lw    $5, 24($29)
    jal   1048, 812
    addiu $4, $4, 1072
    lw    $31, 20($29)
    jr    $31
    move  $2, $0

    subu  $sp, $sp, 32
    sw    $ra, 20($sp)
    sd    $a0, 32($sp)
    sw    $0, 24($sp)
    sw    $0, 28($sp)

loop:
    lw    $t6, 28($sp)
    mul   $t7, $t6, $t6
    lw    $t8, 24($sp)
    addu  $t9, $t8, $t7
    sw    $t9, 24($sp)
    addu  $t0, $t6, 1
    sw    $t0, 28($sp)
    ble   $t0, 100, loop
    la    $a0, str
    lw    $a1, 24($sp)
    jal   printf
    move  $v0, $0
    lw    $ra, 20($sp)
    addu  $sp, $sp, 32
    j     $ra

.data
    .align 0
str:
    .ascii "The sum from 0 .. 100 is %d\n"
  
```

At this point, record the Address of “loop” and used when it is referred

At this point, “str” is not known, make it a blank and filled later when it becomes known.
 => 1-pass is not enough
 => Some can only be resolved from other file

How to Assemble : Labels

Labels stand for the memory address of their definition.

- Computing addresses difficult.
Forward references.
References to other sections.
- Computing all addresses impossible.
References to other files (external).
Only possible during linking.
- Build symbol table to describe label definition & uses.

file1.s:

```
label1:  ...
         beq label2
         ...
label2:  ...
         call foo
         ...
         ba label1
```

file2.s:

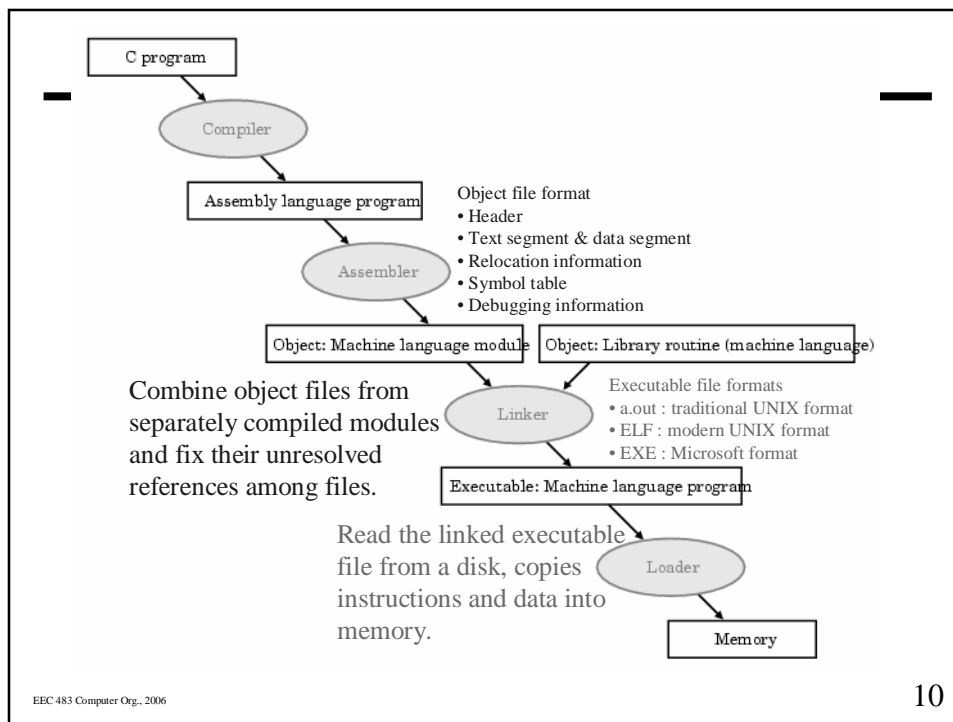
```
foo:  ...
```

How to Resolve Addresses : 2-pass Assembler (& Linker)

- **Pass 1:**
 - Read file & parse input.
 - Build symbol table of where each label defined.
 - Compute starting addresses of each section.
- **Pass 2:**
 - Read file & parse input.
 - Encode, using label's addresses where needed.
 - Write encodings to file.
- Easy to understand but Parses input twice.

How to Resolve Addresses : 1-pass Assembler (& Linker)

- Pass 1:
 - Read & parse input.
 - Encode instructions, with “holes” for labels.
 - Build symbol table of label definitions & uses.
 - Compute starting address of each section.
- Use symbol tables to finish encoding instructions -- “backpatching”.
- Write encodings to file.
- Simpler, and faster (backpatching is faster than a 2nd pass), but, somewhat tricky to backpatch many instruction formats
- Some architecture has 16-bit address field & uses a register if exceeded



Outline of a.out Format

- **Header** Description of rest of file, including segment sizes
- **Segments** Initial contents of memory
 - **Text** Instructions
 - **Rodata** Read-only data, e.g., constants & output strings
 - **Data** Global variables, possibly initialized
 - **Bss** “heap”; pre-allocated space; initialized to zeros
- **Symbol Table** Label definitions & uses (optional for executable files)

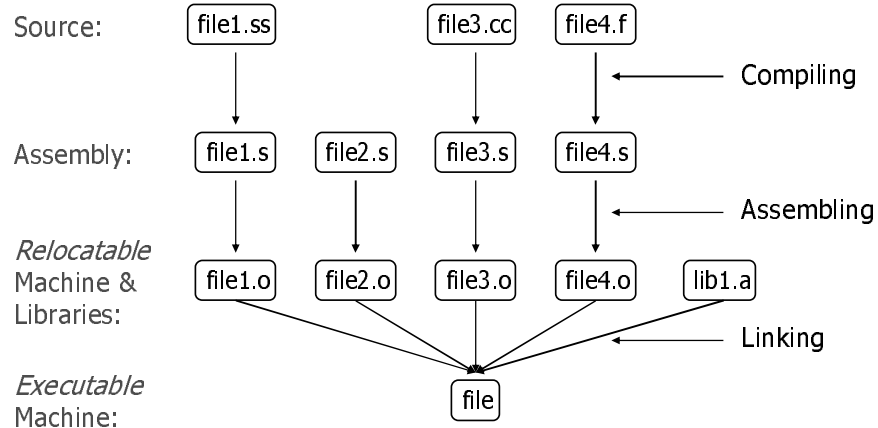
Outline of ELF & EXE Formats

- **Similar to a.out, but much more flexible.**
 - **More kinds of segments available.**
 - **Can specify more information about each segment.**

 - **Lots more details.**

 - **Both simplify shared libraries & dynamic linking.**

Compilation with Multiple Files



Assembling & Linking

Assembling:

- Encode instructions.
- Build table of label definitions.
- Build table of label uses.
- Output relocatable file.

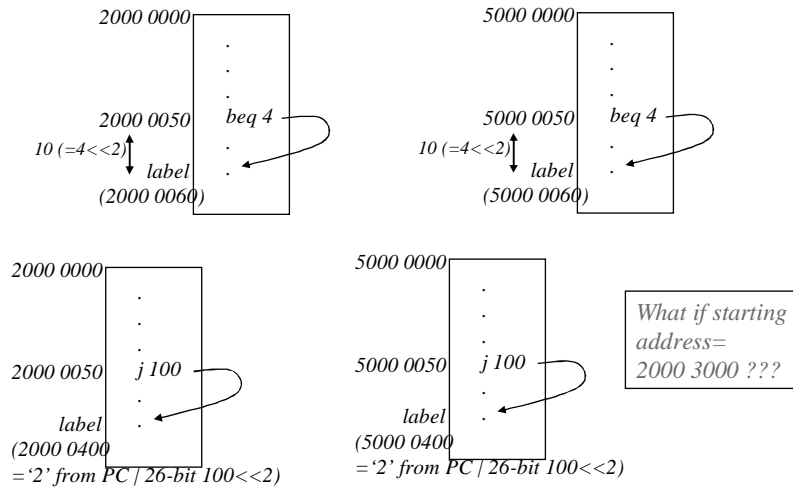
- Encoding is straightforward
- Most of the work is dealing with labels
 - Forward reference : when resolved?
 - Backward reference : when resolved?
 - External reference : when resolved?

Linking:

- "Glue" relocatable files together.
- Assign addresses to labels.
- Replace label uses with addresses.
- Output executable file.

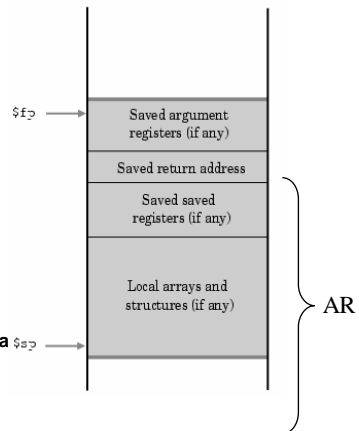
- Details depend on file formats
 - a.out, ELF, EXE
- Next step is "Loader"

Loader & PC-relative addressing



More on Stack (revisited)

- **Procedure frame = activation record**
 - Segment of stack which has a procedure's saved registers and local variables
- **Frame pointer (\$fp)**
 - Is used to reference a local variable stored in stack
 - Why \$sp does not do that ?
- **Global pointer (\$gp)**
 - Is used to reference a static variable stored on data section



2.13 Example: swap => how about “test&set”

```

swap (int v[ ], int k)      // $a0 has starting address of an array v
{                          // $a1 has 'k'
    int temp;

    temp = v[k];           // let's use $t1 for 'temp'
    v[k] = v[k+1];        // $t1 <- memory[$a0+$a1] ???
    v[k+1] = temp;        //
}                          // jr $ra
                          // since return address is in $ra
    
```

```

test&set (int x)
{
    if (x != 1) x=1;
}
Address of "x" is $s0
Use $zero register
    
```

```

Lw $t0, 0($s0)
Addi $t1, $zero, 1
Beq $t0,$t1,exit
Sw $t1, 0($s0)
Exit:
    
```

See our account & draw \$100
=> A: see & draw => B: see & "not" draw
=> A:see, B:see, A:draw, B:draw ???
(multiprogramming,
interleaving with many programs)
=> "Lock"=> A:lock(1), see, draw, unlock(0)
But "Lock" must be "atomic", not a sequence
of multiple operations
=>Implement Test&Set in one machine inst.
=>Semaphore, concurrency control, dining phi

Example: clear1 => how about “make-it-black”

```

clear1 (int array[ ], int size) // $a0 = start address of 'array'
{                               // $a1 = value of 'size'
    int i;                      // $t0 is assigned to local variable 'i'

    for (i = 0; i < size; i = i + 1) // $t0 <- 0
                                    // if $t0 is less than $a1, get out of loop
                                    // (slt)
        array [i] = 0;           // memory[$a0+$t0*4] <- 0
                                    // $t0 <- $t0+1 & repeat the loop
}
    
```

```

make-it-black (char screen[], int size)
{
    for (i=0; i<size; i++)
        screen [i] = 0ffh;
}
Address of screen[] starts at $s2, size in $s3
    
```

```

Sub $t1, $t1, $t1 ; $t1=0 (index)
Loop:
Addiu $t0, $zero, 0ffh ; $t0=0ffh => outside the loop
sw $t0,$t1($s2) ; [screen[0]+i]=ff => Add $t3, $t1, $s2
Sw $t0, 0($t3)                      Sw $t0, 0($t3)
addi $t1, $t1, 4                      => must be 1 (char)
Beq $t1, $s3, exit ; if i=size, exit => it's OK, but must
                                        be just after "loop"
J loop
Exit:
    
```

Example: clear1

```

clear1 (int array[ ], int size)    // $a0 = start address of 'array'
{                                  // $a1 = value of 'size'
    int i;                          // $t0 is assigned to local variable 'i'

    for (i = 0; i < size; i = i + 1) // $t0 <- 0
                                    // if $t0 is less than $a1, get out of loop
                                    //      (slt)
        array [i] = 0;             // memory[$a0+$t0*4] <- 0
                                    // $t0 <- $t0+1 & repeat the loop
}

```

```

loop:  sub    $t0, $t0, $t0           → what's that ?
      slt    $t3, $t0, $a1          }
      beq    $t3, $zero, get_out_loop } why here ?
      add    $t1, $t0, $t0
      add    $t1, $t1, $t1
      add    $t2, $a0, $t1          → $t2 <- $a0+$t0*4
      sw     $zero, 0($t2)
      addi   $t0, $t0, 1
      j     loop
get_out_loop:

```

Example: clear2

```

clear2 (int *array, int size)    // $a0 = start address of 'array'
{                                  // $a1 = value of 'size'
    int *p;                          // $t0 is assigned to local variable 'p'

    for (p = &array[0]; p < &array[size]; p = p + 1)
                                    // compare with "for (i=0; i<size; i=i+1)"
                                    // $t0 <- $a0
                                    // if $t0 is less than $a0+size*4, get out
                                    //      (slt)
        *p = 0;                     // memory[$t0] <- 0
                                    // $t0 <- $t0+1*4 & repeat the loop
}

```

*p = 0;

address of
array[0]

in fact, increment
by four

Example: clear2

```
clear2 (int *array, int size)    // $a0 = start address of 'array'
{
    int *p;                      // $a1 = value of 'size'
    for (p = &array[0]; p < &array[size]; p = p + 1) // $t0 <- $a0
        // if $t0 is less than $a0+size*4, get out
        // (slt)
        *p = 0;                  // memory[$t0] <- 0
        // $t0 <- $t0+1*4 & repeat the loop
}
```

```

    add    $t0, $a0, $zero
    add    $t1, $a1, $a1
    add    $t1, $t1, $t1
    add    $t2, $a0, $t1
loop:     slt    $t3, $t0, $t2
        beq    $t3, $zero, get_out_loop
        sw    $zero, 0($t0)
        addi   $t0, $t0, 4
        j     loop
get_out_loop:
```

→ \$t2 <- \$a0+\$a1*4

2.15 Array versus Pointer

```
clear1 (int array[ ], int size)
{
    int i;

    for (i = 0; i < size; i = i + 1)
        array [i] = 0;
}

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

Which one is better ?

Comparison : Array versus Pointer

```

loop:  sub    $t0, $t0, $t0          add    $t0, $a0, $zero
      slt   $t3, $t0, $a1       add    $t1, $a1, $a1
      beq   $t3, $zero, get_out_loop  add    $t1, $t1, $t1
      add   $t1, $t0, $t0       add    $t2, $a0, $t1
      add   $t1, $t1, $t1       loop:  slt   $t3, $t0, $t2
      add   $t2, $a0, $t1       beq   $t3, $zero, get_out_loop
      sw    $zero, 0($t2)       sw    $zero, 0($t0)
      addi  $t0, $t0, 1         addi  $t0, $t0, 4
      j     loop                j     loop
get_out_loop:                    get_out_loop:

```

Which one is better ?

2.16 Alternative Architectures

- **Design alternative:**
 - provide more powerful operations
 - goal is to reduce number of instructions executed
 - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
 - virtually all new instruction sets since 1982 have been RISC
 - VAX: minimize code size, make assembly language easy
instructions from 1 to 54 bytes long!
- We'll look at PowerPC and 80x86
- And then IA-32

IA - 32

- 1978: The Intel 8086 is announced (16 bit architecture)
 - 1980: The 8087 floating point coprocessor is added
 - 1982: The 80286 increases address space to 24 bits, +instructions
 - 1985: The 80386 extends to 32 bits, new addressing modes
 - 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
 - 1997: 57 new “MMX” instructions are added, Pentium II
 - 1999: The Pentium III added another 70 instructions (SSE)
 - 2001: Another 144 instructions (SSE2)
 - 2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
 - 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions
- “This history illustrates the impact of the “golden handcuffs” of compatibility
“adding new features as someone might add clothing to a packed bag”
“an architecture that is difficult to explain and impossible to love”

IA-32 Overview

- Complexity:
 - Instructions from 1 to 17 bytes long
 - one operand must act as both a source and destination
 - one operand can come from memory
 - complex addressing modes
e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
 - the most frequently used instructions are not too difficult to build
 - compilers avoid the portions of the architecture that are slow

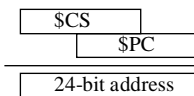
*“what the 80x86 lacks in style is made up in quantity,
making it beautiful from the right perspective”*

A dominant architecture: 80x86

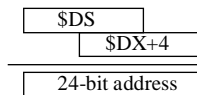
- Complexity:
 - Instructions from 1 to 17 bytes long ??? “irregular”
 - one operand must act as both a source and destination ???
 - one operand can come from memory ???
 - complex addressing modes “irregular”
e.g., “base or scaled index with 8 or 32 bit displacement”

Until 80286...

Next address = \$CS // \$PC



lw \$AX, 4(\$DX)



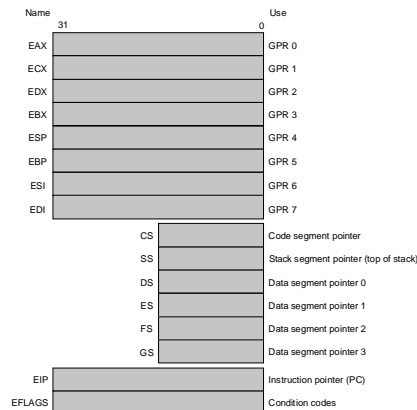
Why ?

\$CS: code segment
\$DS: data segment
\$SS: stack segment
\$SI: source index
\$DI: destination index

Not really
a GPR architecture !

IA-32 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386



IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	lw \$s0,0(\$s1)
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	lw \$s0,100(\$s1)# ≤16-bit displacement
Base plus scaled Index	The address is $Base + (2^{Scale} \times Index)$ where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,0(\$t0)
Base plus scaled Index with 8- or 32-bit displacement	The address is $Base + (2^{Scale} \times Index) + displacement$ where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,100(\$t0)# ≤16-bit displacement

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a lui to load the upper 16 bits of the displacement and an add to sum the upper address with the base register \$s1. (Intel gives two different names to what is called based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

IA-32 Typical Instructions

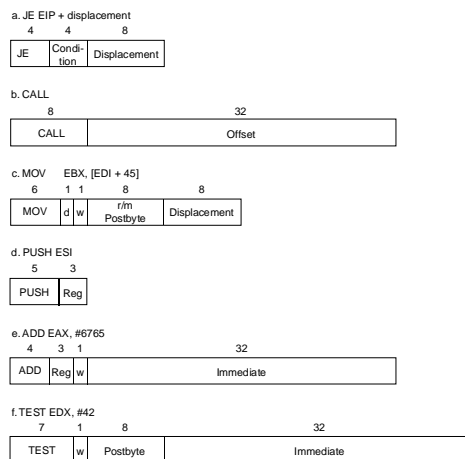
- Four major types of integer instructions:
 - Data movement including move, push, pop
 - Arithmetic and logical (destination register or memory)
 - Control flow (use of condition codes / flags)
 - String instructions, including string move and string compare

Instruction	Function
JE name	If equal (condition code) [EIP=name]; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOV EBX, [EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX, #6765	EAX=EAX+6765
TEST EDX, #42	Set condition code (flags) with EDX and 42
MOVS	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

IA-32 instruction Formats

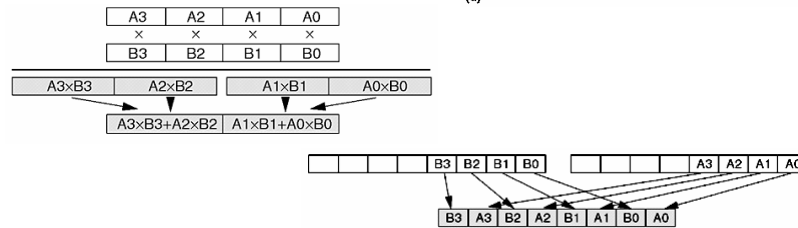
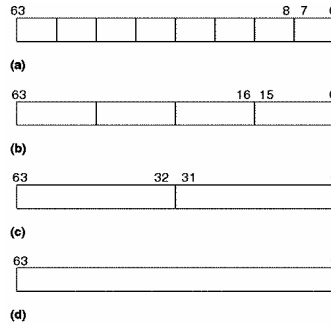
- Typical formats: (notice the different lengths)



Intel MMX

- **Packed data type**

- packed byte, 8 bytes packed into one 64-bit quantity (a),
- packed word, 4 words packed into one 64-bit quantity (b),
- packed doubleword, 2 doublewords packed into one 64-bit quantity (c), and
- quadword, one 64-bit quantity (d).



Intel MMX

Table 1. MMX instruction set summary.

Opcode mnemonic*	Options	Cycle count	Description
PADD(<i>b,w,d</i>)	Wraparound	1	Adds or subtracts packed 8 bytes, four 16-bit words, or two
PSUB(<i>b,w,d</i>)	and saturate		32-bit doublewords in parallel.
PCMPSEQ(<i>b,w,d</i>)	Equal or greater than	1	Compares packed 8 bytes, four 16-bit words, or two 32-bit
PCMPGT(<i>b,w,d</i>)			elements in parallel. Result is mask of 1s if true or 0s if false.
PMULLW	Result high-	Latency: 3,	Multiplies four packed, signed 16-bit words in parallel.
PMULHW	or low-order bits	throughput: 1	Chooses low- or high-order 16 bits of the 32-bit result.
PMADDWD	Word to doubleword	Latency: 3,	Multiplies four packed, signed 16-bit words and adds
conversion	throughput: 1	together adjacent pairs of 32-bit results in parallel. Result	is a doubleword.
PSRA(<i>w,d</i>), PSLL(<i>w,d,q</i>)	Shift count in register	1	Shifts arithmetic right, logical left and right packed 4 words,
PSRL(<i>w,d,q</i>)	or immediate		2 doublewords, or the full 64 bits (quadword) in parallel.
PUNPCKL(<i>bw,wd,dd</i>)	—	1	Merges packed 8 bytes, four 16-bit words, or two