Can you see why it make sense for the parameters to be stacked below (before) the return address?
Quiz on Ch.10 (Flags and codes pp.378-9)

- Why does the x86 instruction set have two classes of comparison codes?
  - G, L, GE, LE,
  - A (above), B (below), AE, BE

- Explain the following line of the table 10.10/379:

<table>
<thead>
<tr>
<th>G, NLE</th>
<th>[(S=1 AND O=1) OR (S=0 and O=0)] AND [Z=0]</th>
<th>Greater than; Not less than or equal (signed)</th>
</tr>
</thead>
</table>

Chapter 11
Instruction Sets:
Addressing Modes and Formats
Addressing Modes

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement (Indexed)
- Stack

What is Effective Address (EA)?
## Immediate Addressing Diagram

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand</th>
</tr>
</thead>
</table>

**Instruction**
Immediate Addressing

- Operand is part of instruction
- Operand = operand field
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is the operand
- No memory reference to access data
- Fast
- Range of operands limited to # of bits in operand field (< word size)
Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Range of addresses limited by # of bits in A (< word length)
Indirect Addressing Diagram

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Address A</th>
</tr>
</thead>
</table>

Instruction

Memory

Pointer to operand

Operand
Indirect Addressing

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- $EA = (A)$
  - Look in A, find address $(A)$ and look there for operand
- e.g. ADD $(A)$
  - Add contents of cell pointed to by contents of A to accumulator
Indirect Addressing

- Advantage: Large address space
  - $2^n$ where $n =$ word length
- Disadvantage: Multiple memory accesses to find operand $\rightarrow$ slower

- May be nested, multilevel, cascaded
  - e.g. $EA = (((A)))$
    - Draw the diagram!
### Register Addressing Diagram

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Register Address R</th>
</tr>
</thead>
</table>

Registers, in CPU

Operand
Register Addressing

- Operand is held in register named in address filed
- EA = R

Advantages:
- Very small address field
  - Shorter instructions
  - Faster instruction fetch
- Faster memory access to operand(s)
Register Addressing

Disadvantage:

- Very limited address space

- Multiple registers helps performance
  - Requires good assembly programming or compiler writing
  - C language has a dedicated keyword:

  ```
  register int a;
  ```
Register Indirect Addressing Diagram

Instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Register Address R</th>
</tr>
</thead>
</table>

Registers, in CPU

Pointer to Operand

Memory

Operand
Register Indirect Addressing

- C.f. indirect addressing
- EA = (R)
- Operand is in memory cell pointed to by contents of register R

Comparison with (memory) indirect:
- Same large address space \(2^n\)
- One less memory access!
Displacement Addressing Diagram

Instruction

OpCode  Register R  Address A

Registers

Pointer to Operand

MEMORY

Memory

Operand
Displacement Addressing

- $EA = A + (R)$
- Address field holds two values
  - $A = \text{base value}$
  - $R = \text{register that holds displacement}$
  - or vice versa
- Has many versions, of which we mention these 3:
  - Relative
  - Base-register
  - Indexing
Relative (to PC) Addressing

It’s a version of displacement addressing

- \( R = \) Program counter, PC
- \( EA = A + (PC) \)
  - The operand is \( A \) cells away from the current cell (the one pointed to by PC)

- Remember:
  - locality of reference
  - cache usage
Base-Register Addressing

It’s a version of displacement addressing
It’s a generalized relative addressing, where other registers can play the role of PC

- A holds displacement
- R holds pointer to base address
- EA = A + (R)
  - R may be explicit or implicit
- E.g. six segment registers in 80x86: CS, DS, ES, FS, GS, SS
Indexed Addressing

It’s a version of displacement addressing
Very similar to base-register addressing

- $A = \text{base}$
- $(R) = \text{displacement}$
- $EA = A + (R)$, but roles are reversed!
- Good for accessing arrays

$R++$

- **Autoindexing**: the incrementations/decrementation is performed in the same instruction cycle!
Combinations

• Postindex: \( EA = (A) + (R) \)
  - First use \( A \) as direct addressing \( \rightarrow \) at address \( A \) we find an address \( A1 \).
  - “Index” \( A1 \) based on \( R \) \( \rightarrow \) add \( A1 \) to the content of \( R \)

• Preindex: \( EA = (A+(R)) \)

Draw the diagrams!
Stack Addressing

- Operand is (implicitly) on top of the stack
- E.g. ADD
  - Pop top two items from stack
  - Add them
  - Push result on top of stack
Review:

What addressing mode is used in the x86 instruction

`MOV EAX 42`
Review:

What addressing modes were used in the IAS instruction set?
See next slides ...
# The IAS instruction set

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Opcode</th>
<th>Symbolic Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data transfer</strong></td>
<td>00001010</td>
<td>LOAD MQ</td>
<td>Transfer contents of register MQ to the accumulator AC</td>
</tr>
<tr>
<td></td>
<td>00001001</td>
<td>LOAD MQ,M(X)</td>
<td>Transfer contents of memory location X to MQ</td>
</tr>
<tr>
<td></td>
<td>00100001</td>
<td>STOR M(X)</td>
<td>Transfer contents of accumulator to memory location X</td>
</tr>
<tr>
<td></td>
<td>00000001</td>
<td>LOAD M(X)</td>
<td>Transfer M(X) to the accumulator</td>
</tr>
<tr>
<td></td>
<td>00000010</td>
<td>LOAD –M(X)</td>
<td>Transfer –M(X) to the accumulator</td>
</tr>
<tr>
<td></td>
<td>00000011</td>
<td>LOAD</td>
<td>M(X)</td>
</tr>
<tr>
<td></td>
<td>00000100</td>
<td>LOAD –</td>
<td>M(X)</td>
</tr>
<tr>
<td><strong>Unconditional branch</strong></td>
<td>00001101</td>
<td>JUMP M(X,0:19)</td>
<td>Take next instruction from left half of M(X)</td>
</tr>
<tr>
<td></td>
<td>00001110</td>
<td>JUMP M(X,20:39)</td>
<td>Take next instruction from right half of M(X)</td>
</tr>
<tr>
<td><strong>Conditional branch</strong></td>
<td>00001111</td>
<td>JUMP+ M(X,0:19)</td>
<td>If number in the accumulator is nonnegative, take next instruction from left half of M(X)</td>
</tr>
<tr>
<td></td>
<td>00100000</td>
<td>JUMP+ M(X,20:39)</td>
<td>If number in the accumulator is nonnegative, take next instruction from right half of M(X)</td>
</tr>
</tbody>
</table>

Specifies one of 21 instructions

There was no assembly language back then!
### IAS – instruction set (continued)

<table>
<thead>
<tr>
<th>Address modify</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010010</td>
<td>STOR M(X,8:19)</td>
<td>Replace left address field at M(X) by 12 rightmost bits of AC</td>
</tr>
<tr>
<td>00010011</td>
<td>STOR M(X,28:39)</td>
<td>Replace right address field at M(X) by 12 rightmost bits of AC</td>
</tr>
</tbody>
</table>

Self-modification of code, but it was done to “simulate” today’s indirect and displacement addressing.
We covered section 11.1 of the text. Please read carefully, it’s very important!

Solve in notebook end-of-chapter:

- Review questions 1-11
- Problem 1
Detour: Ch.8, Section 8.4:

Two ways to subdivide physical memory in Intel x86: paging and segmentation

- Paging is invisible to the programmer. We cover it in CS 380 (Operating Systems)
- Segmentation is usually visible to the programmer. It provides:
  - Convenience for organizing programs and data
  - A means to implement privilege and protection mechanisms for processes
  - Help for identifying bugs during program development
Two ways to subdivide physical memory: paging and segmentation

- Pentium II includes hardware for both paging and segmentation. Each mechanism can be enabled separately, resulting in 4 modes:
  - unsegmented unpaged
  - unsegmented paged a.k.a. page-protected
  - segmented unpaged a.k.a. segment-protected
  - segmented paged a.k.a. protected
Pentium II Complete Address Translation Mechanism (segmentation + paging)

When paging is not used, the linear address is placed directly on the memory bus.
**Pentium II Segmentation**

- Each virtual address is 16-bit segment and 32-bit offset
- 2 bits of segment are protection mechanism
- 14 bits specify segment
- Unsegmented virtual memory $2^{32} = 4\text{Gbytes}$
- Segmented $2^{46} = 64 \text{ terabytes}$
  - Can be larger – depends on which process is active
  - Half (8K segments of 4Gbytes) is global
  - Half is local and distinct for each process
Pentium II Segment Selector + Table

Segment selector = 2 Byte
Segment descriptor = 8 Byte

Source for images: http://users.evtek.fi/~tk/rtos/TSS/protmode.htm
Pentium II Protection

- Protection bits give 4 levels of privilege
  - 0 most protected, 3 least
  - Use of levels software dependent
  - Usually level 3 for applications, level 1 for O/S and level 0 for kernel (level 2 not used)
  - Level 2 may be used for apps that have internal security e.g. database
  - Some instructions only work in level 0 (used for OS for memory management)
Pentium II Address Translation – Segmentation only

Source for image: http://users.evtek.fi/~tk/rtos/TSS/protmode.htm

End Ch. 8
11.2 x86 Addressing Modes

The effective address (EA in section 11.1) is the virtual address, and used as offset into a segment

- Starting address plus offset gives linear address
- This then goes through page translation if paging is enabled
x86 Addressing Mode Calculation
Which segment register is used?
Determined by:
• instruction itself (e.g. stack-related instructions use SS)
• context of execution (e.g. in a multi-process environment, a certain process is assigned a certain Segment Register by the OS)
x86 Addressing Modes

12 addressing modes:
- Immediate (Byte, word, doubleword)
- Register operand (8, 16, 32 and 64-bit registers)
- Displacement
- Base
- Base with displacement
- Scaled index with displacement
- Base with index and displacement
- Base scaled index with displacement
- Relative

Operands in memory
x86 Registers

Source for image: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html x86 Assembly Guide
**x86 Addressing Modes – operand in memory**

A segment register is used for all except relative (PC)

- Displacement: EA contained in instruction
  - 8, 16 or 32 bit
  - Can lead to long instructions, esp. for 32 bit!
  - Can be used to reference global variables

- Base
- Base with displacement
- Scaled index with displacement
- Base with index and displacement
- Base scaled index with displacement
- Relative

(indirect addressing)
## x86 Addressing Modes

<table>
<thead>
<tr>
<th>Type</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Operand = A</td>
</tr>
<tr>
<td>Register Operand</td>
<td>LA = R</td>
</tr>
<tr>
<td>Displacement</td>
<td>LA = (SR) + A</td>
</tr>
<tr>
<td>Base</td>
<td>LA = (SR) + (B)</td>
</tr>
<tr>
<td>Base with Displacement</td>
<td>LA = (SR) + (B) + A</td>
</tr>
<tr>
<td>Scaled Index with Displacement</td>
<td>LA = (SR) + (I) × S + A</td>
</tr>
<tr>
<td>Base with Index and Displacement</td>
<td>LA = (SR) + (B) + (I) + A</td>
</tr>
<tr>
<td>Base with Scaled Index and Displacement</td>
<td>LA = (SR) + (I) × S + (B) + A</td>
</tr>
<tr>
<td>Relative</td>
<td>LA = (PC) + A</td>
</tr>
</tbody>
</table>

- **LA** = linear address
- **(X)** = contents of X
- **SR** = segment register
- **PC** = program counter
- **A** = contents of an address field in the instruction
- **R** = register
- **B** = base register
- **I** = index register
- **S** = scaling factor
SKIP ARM Addressing Modes
We covered sections 8.4 (Only segmentation, not paging) and 11.2 of the text. Please read carefully, it’s very important!

Solve in notebook end-of-chapter 11 problems:

- 2, 3, 4, 8
11.3 Instruction Formats

- Format = Layout of bits in an instruction
- Includes opcode
- Includes (implicit or explicit) operand(s)
  - Zero or more
- Usually more than one instruction format in an instruction set
Instruction Length

- Affected by and affects:
  - Memory size
  - Memory organization
  - Bus structure
  - CPU complexity
  - CPU speed

- Trade-off between:
  - Power of instructions
    - Many opcodes, operands, addressing modes
    - Greater address range
  - Saving space
    - Amount of memory used to store a program
    - # of fetch and get cycles
Instruction Length

• Granularity:
  — Instruction length should be an integer multiple of the word length
    – Data types
    – Bus width
  — “No more 20-bit instructions, Dr. von Neumann!” 😊
  — IBM System/360 started using 8-bit characters (EBCDIC!) and simultaneously the Byte-addressable memory
    – As opposed to the 7000 series, which had been accessing memory chunks of variable size at arbitrary bit addresses
Allocation of Bits

Determines:

- Number of addressing modes
- Number of operands
- Register versus memory
- Number of register sets, e.g. for x86:
  - General-purpose
  - Data (floating-point, MMX)
  - Displacement (segment reg.)
- Address range
- Address granularity: bytes or words?
Allocation of Bits

Again, numerous trade-offs, e.g.

- # of opcodes vs. # of addresses (for a fixed-length instruction)
- Variable-length opcodes, but:
  - Need more hardware
  - Need “prefix code” – see next slide
PEP Instructions have variable opcode length
(not in our text)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Stop execution</td>
</tr>
<tr>
<td>1100</td>
<td>Load the operand into the A register</td>
</tr>
<tr>
<td>1110</td>
<td>Store the contents of the A register into the operand</td>
</tr>
<tr>
<td>0111</td>
<td>Add the operand to the A register</td>
</tr>
<tr>
<td>1000</td>
<td>Subtract the operand to the A register</td>
</tr>
<tr>
<td>01001</td>
<td>Character input to the operand</td>
</tr>
<tr>
<td>01010</td>
<td>Character output from the operand</td>
</tr>
</tbody>
</table>
Allocation of Bits

Another trade-off:

- # of register sets
- E.g. 64 “flat” registers vs. four sets of 16
  → 6 bits vs. 4
- Can you think of a disadvantage?

Read the entire p.415 carefully!
Orthogonality

A computer's instruction set is said to be **orthogonal** if any instruction can use data of any type via any addressing mode.

Idea: Specify the addressing mode in the operand, rather than the opcode.

Advantage ... Disadvantage ...

The DEC PDP-11 and Motorola 68000 computer architectures are examples of nearly orthogonal instruction sets, while the ARM11 and VAX are examples of CPUs with fully orthogonal instruction sets. [Source: Orthogonal instruction set - Wikipedia]
Text reading assignment:

The remainder of section 11.3 (pp.416-421)

Take notes only about orthogonality!
11.4 x86 Instruction Format

- Total Byte-long prefixes: 0 to 4
- Optional address specifier
Two functions:

- **LOCK** → restricts other instructions’ use of shared memory
- **REPEAT** → repeated operation on a string, e.g.
  - **REP** means that the operation is performed a number of times specified by register CX (C on 16 bit)
LOCK prefix

• Only meaningful in multi-threading and multi-processor applications

• Can be used only with the following instructions (and to those forms of the instructions that use a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.

• An undefined opcode exception will be generated if the LOCK prefix is used with any other instruction.

• The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

Hardware signal in the Intel CPU
X86 Instruction Format

Operand is in register or memory?
Which register?
Read and take notes: pp.422-423

SKIP ARM Instruction Formats
Quiz on instruction formats

Problem 11.16/431

• Hint: Assume that there are no 3- or more-operand instructions
11.5 What does the assembler do?

Computers store and “understand” binary instructions, a.k.a. machine code

Example: Calculate \( N = I + J + K \)

- Program starts in memory location 101 (hex)
- Data starts at 201 (hex)
- Code:
  - Load contents of 201 into AC
  - Add contents of 202 to AC
  - Add contents of 203 to AC
  - Store contents of AC to 204

- See machine code on next slide
Program in binary  
(Fig. 11.3(a) is messed up in text!)

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0010</td>
</tr>
<tr>
<td>102</td>
<td>0001</td>
</tr>
<tr>
<td>103</td>
<td>0001</td>
</tr>
<tr>
<td>104</td>
<td>0011</td>
</tr>
<tr>
<td>..........</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>0000</td>
</tr>
<tr>
<td>202</td>
<td>0000</td>
</tr>
<tr>
<td>203</td>
<td>0000</td>
</tr>
<tr>
<td>204</td>
<td>0000</td>
</tr>
</tbody>
</table>

Do you see the opcodes?

How many bits/Bytes are there at any memory address?
## Improvement: convert binary to hex

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0010 0010 0000 0001</td>
</tr>
<tr>
<td>102</td>
<td>0001 0010 0000 0010</td>
</tr>
<tr>
<td>103</td>
<td>0001 0010 0000 0011</td>
</tr>
<tr>
<td>104</td>
<td>0011 0010 0000 0100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>201</td>
<td>0000 0000 0000 0010</td>
</tr>
<tr>
<td>202</td>
<td>0000 0000 0000 0011</td>
</tr>
<tr>
<td>203</td>
<td>0000 0000 0000 0100</td>
</tr>
<tr>
<td>204</td>
<td>0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>2201</td>
</tr>
<tr>
<td>102</td>
<td>1202</td>
</tr>
<tr>
<td>103</td>
<td>1203</td>
</tr>
<tr>
<td>104</td>
<td>3204</td>
</tr>
<tr>
<td>201</td>
<td>0002</td>
</tr>
<tr>
<td>202</td>
<td>0003</td>
</tr>
<tr>
<td>203</td>
<td>0004</td>
</tr>
<tr>
<td>204</td>
<td>0000</td>
</tr>
</tbody>
</table>
Use symbolic names (a.k.a. mnemonics) for instructions

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>LDA</td>
<td>201</td>
</tr>
<tr>
<td>102</td>
<td>ADD</td>
<td>202</td>
</tr>
<tr>
<td>103</td>
<td>ADD</td>
<td>203</td>
</tr>
<tr>
<td>104</td>
<td>STA</td>
<td>204</td>
</tr>
<tr>
<td>201</td>
<td>DAT</td>
<td>2</td>
</tr>
<tr>
<td>202</td>
<td>DAT</td>
<td>3</td>
</tr>
<tr>
<td>203</td>
<td>DAT</td>
<td>4</td>
</tr>
<tr>
<td>204</td>
<td>DAT</td>
<td>0</td>
</tr>
</tbody>
</table>

Rules:
- Three fields per line
- Location address
- Three letter opcode
- If memory reference: use address in hex

Problem!
## Symbolic Addresses!

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMUL</td>
<td>LDA</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>J</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>K</td>
</tr>
<tr>
<td></td>
<td>STA</td>
<td>N</td>
</tr>
<tr>
<td>I</td>
<td>DATA</td>
<td>2</td>
</tr>
<tr>
<td>J</td>
<td>DATA</td>
<td>3</td>
</tr>
<tr>
<td>K</td>
<td>DATA</td>
<td>4</td>
</tr>
<tr>
<td>N</td>
<td>DATA</td>
<td>0</td>
</tr>
</tbody>
</table>
• Need an assembler to translate from assembly to machine code
• Assembler are still used for some systems programming:
  — Compilers
  — I/O routines
Homework for Ch.11
Due Thu, Nov 25

End-of-chapter 11 problems:

- 6
- 7
- 11
- 13
- 20