Lecture 03: ISA Introduction

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Contents

1. Introduction
2. Classifying Instruction Set Architectures
3. Memory Addressing
4. Type and Size of Operands
5. Operations in the Instruction Set
6. Instructions for Control Flow
7. Encoding an Instruction Set
8. Crosscutting Issues: The Role of Compilers
9. RISC-V ISA

• Supplement
  – MIPS ISA
  – RISC vs CISC
  – Compiler compilation stages
  – ISA Historical
    • Appendix L
  – Comparison of ISA
    • Appendix K
The MIPS Instruction Set

• Used as the example as introduction
• Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
• Large share of embedded core market
  – Applications in consumer electronics, network/storage equipment, cameras, printers, …
  – Closed one to RISC-V
• Typical of many modern ISAs
  – See MIPS Reference Data tear-out card, and Appendixes B and E in the reference textbook
Arithmetic Operations

• Add and subtract, three operands
  – Two sources and one destination
    \[
    \text{add } a, b, c \quad \# \quad a \gets b + c
    \]
• 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 MIPS 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
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations
Register Operand Example

• C code:
  \[ f = (g + h) - (i + j); \]
  - \( f, \ldots, j \) in \( $s0, \ldots, $s4 \)

• Compiled MIPS code:
  \[
  \begin{align*}
  &\text{add } $t0, $s1, $s2 \\
  &\text{add } $t1, $s3, $s4 \\
  &\text{sub } $s0, $t0, $t1
  \end{align*}
  \]
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
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - c.f. Little Endian: least-significant byte at least address
Memory Operand Example 1

- C code:
\[
g = h + A[8];
\]
- g in $s1, h in $s2, **base address** of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32
  - **4 bytes per word**
  
  \[
  \text{lw } \$t0, 32($s3) \quad \# \text{ load word}
  \]
  
  \[
  \text{add } \$s1, \$s2, \$t0
  \]
Memory Operand Example 2

• C code:
  \[ A[12] = h + A[8]; \]
  – h in $s2, base address of A in $s3

• Compiled MIPS code:
  – Index 8 requires offset of 32
  
  \[
  \begin{align*}
  \text{lw} & \quad \text{$t0}, \quad 32($s3) \quad \text{# load word} \\
  \text{add} & \quad \text{$t0}, \quad $s2, \quad $t0 \\
  \text{sw} & \quad \text{$t0}, \quad 48($s3) \quad \text{# store word}
  \end{align*}
  \]
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
  \texttt{addi \$s3, \$s3, 4}

• No subtract immediate instruction
  – Just use a negative constant
    \texttt{addi \$s2, \$s1, -1}

• \textit{Design Principle 3}: Make the common case fast
  – Small constants are common
  – Immediate operand avoids a load instruction
The Constant Zero

• MIPS/RISC-V register 0 ($zero) is the constant 0
  – Cannot be overwritten

• Useful for common operations
  – E.g., move between registers
    add $t2, $s1, $zero
Representing Instructions

• Instructions are encoded in binary
  – Called machine code

• MIPS instructions
  – Encoded as 32-bit instruction words
  – Small number of formats encoding operation code (opcode), register numbers, …
  – Regularity!

• Register numbers
  – $t0 – t7$ are reg’s 8 – 15
  – $t8 – t9$ are reg’s 24 – 25
  – $s0 – s7$ are reg’s 16 – 23
### MIPS R-format Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **Instruction fields**
  - **op**: operation code (opcode)
  - **rs**: first source register number
  - **rt**: second source register number
  - **rd**: destination register number
  - **shamt**: shift amount (00000 for now)
  - **funct**: function code (extends opcode)
**R-format Example**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

`add $t0, $s1, $s2`

<table>
<thead>
<tr>
<th>special</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

\[0000001000110100100000000100000_2 = 02324020_{16}\]
Hexadecimal

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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
<td>1000</td>
<td>c</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>d</td>
<td>1101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>a</td>
<td>1010</td>
<td>e</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>b</td>
<td>1011</td>
<td>f</td>
<td>1111</td>
</tr>
</tbody>
</table>

Example: eca8 6420
- 1110 1100 1010 1000 0110 0100 0010 0000
MIPS I-format Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **Immediate arithmetic and load/store instructions**
  - rt: destination or source register number
  - Constant: $-2^{15}$ to $+2^{15} - 1$
  - Address: offset added to base address in rs

- **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

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **shamt**: how many positions to shift
- **Shift left logical**
  - Shift left and fill with 0 bits
  - \( \text{sll} \) by \( i \) bits multiplies by \( 2^i \)
- **Shift right logical**
  - Shift right and fill with 0 bits
  - \( \text{srl} \) by \( i \) bits divides by \( 2^i \) (unsigned only)
AND Operations

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

\[
\text{and } t0, \ t1, \ t2
\]
OR Operations

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

or \$t0, \$t1, \$t2

<table>
<thead>
<tr>
<th>$t2</th>
<th>0000 0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0</td>
<td>0000 0000 0000 0000 0000 0011 1101 1100 0000</td>
</tr>
</tbody>
</table>
NOT Operations

• Useful to invert bits in a word
  – Change 0 to 1, and 1 to 0
• MIPS has NOR 3-operand instruction
  – \( a \text{ NOR } b = \text{ NOT ( } a \text{ OR } b ) \)

\[
\text{nor } \$t0, \$t1, \$zero \\
\]

\[
\begin{array}{c}
\$t0 \\
\$t1
\end{array}
\begin{array}{cccccccccccc}
1111 & 1111 & 1111 & 1111 & 1100 & 0011 & 1111 & 1111 \\
0000 & 0000 & 0000 & 0000 & 0011 & 1100 & 0000 & 0000
\end{array}
\]

Register 0: always read as zero
Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- beq rs, rt, L1
  - if (rs == rt) branch to instruction labeled L1;
- bne rs, rt, L1
  - if (rs != rt) branch to instruction labeled L1;
- j L1
  - unconditional jump to instruction labeled L1
Compiling If Statements

- C code:
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```
  - f, g, ... in $s0, $s1, ...

- Compiled MIPS code:
  ```mips
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  ```
  ```mips
  Else:
  sub $s0, $s1, $s2
  ```
  ```mips
  Exit: ...
  ```

Assembler calculates addresses
Compiling Loop Statements

• C code:
  
  ```c
  while (save[i] == k) i += 1;
  ```
  
  – i in $s3, k in $s5, address of save in $s6

• Compiled MIPS code:

  ```mips
  Loop: sll $t1, $s3, 2 /* x4 */  
  add $t1, $t1, $s6  
  lw $t0, 0($t1)  
  bne $t0, $s5, Exit  
  addi $s3, $s3, 1  
  j Loop
  ```

  ```mips
  Exit: ...
  ```
More Conditional Operations

• Set result to 1 if a condition is true
  – Otherwise, set to 0
• `slt rd, rs, rt`
  – `if (rs < rt) rd = 1; else rd = 0;`
• `slti rt, rs, constant`
  – `if (rs < constant) rt = 1; else rt = 0;`
• Use in combination with `beq`, `bne`
  `slt $t0, $s1, $s2  # if ($s1 < $s2)`
  `bne $t0, $zero, L  # branch to L`
Branch Addressing

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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
Jump Addressing

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

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- (Pseudo)Direct jump addressing
  - Target address = \( PC_{31...28} : (\text{address} \times 4) \)
Target Addressing Example

- Loop code from earlier example
  
  Assume Loop at location 80000

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>$t1, $s3, 2</th>
<th>$t1, $t1, $s6</th>
<th>$t0, 0($t1)</th>
<th>$t0, $s5, Exit</th>
<th>$s3, $s3, 1</th>
<th>Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: sll</td>
<td>80000</td>
<td>0</td>
<td>9</td>
<td>19</td>
<td>9</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>add</td>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>lw</td>
<td>80008</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>80012</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>80016</td>
<td>8</td>
<td>19</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>80020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit: ...</td>
<td>80024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Addressing Mode Summary

1. Immediate addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} & \text{Immediate} \\
   \end{array}
   \]

2. Register addressing
   \[
   \begin{array}{cccc}
   \text{op} & \text{rs} & \text{rt} & \text{rd} & \ldots & \text{funct} \\
   \end{array}
   \]

3. Base addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} & \text{Address} \\
   \end{array}
   \]

4. PC-relative addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} & \text{Address} \\
   \end{array}
   \]

5. Pseudodirect addressing
   \[
   \begin{array}{c}
   \text{op} & \text{Address} \\
   \end{array}
   \]
C Sort Example

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

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

```assembly
swap:  sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
      #   (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
```
The Sort Procedure in C

• 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);
        }
    }
}
– v in $a0, k in $a1, i in $s0, j in $s1
The Procedure Body

move $s2, $a0           # save $a0 into $s2
move $s3, $a1           # save $a1 into $s3
move $s0, $zero         # i = 0

for1tst: slt $t0, $s0, $s3 # $t0 = 0 if $s0 \geq $s3 (i \geq n)
beq $t0, $zero, exit1  # go to exit1 if $s0 \geq $s3 (i \geq n)
addi $s1, $s0, -1       # j = i - 1

for2tst: slti $t0, $s1, 0  # $t0 = 1 if $s1 < 0 (j < 0)
bne $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
sll $t1, $s1, 2        # $t1 = j * 4
add $t2, $s2, $t1      # $t2 = v + (j * 4)
lw $t3, 0($t2)        # $t3 = v[j]
lw $t4, 4($t2)        # $t4 = v[j + 1]
slt $t0, $t4, $t3      # $t0 = 0 if $t4 \geq $t3
beq $t0, $zero, exit2  # go to exit2 if $t4 \geq $t3
move $a0, $s2           # 1st param of swap is v (old $a0)
move $a1, $s1           # 2nd param of swap is j
jal swap                # call swap procedure
addi $s1, $s1, -1       # j -= 1
j for2tst               # jump to test of inner loop

exit2: addi $s0, $s0, 1   # i += 1
j for1tst               # jump to test of outer loop
The Full Procedure

```
sort:       addi $sp,$sp, -20  # make room on stack for 5 registers
            sw $ra, 16($sp)  # save $ra on stack
            sw $s3,12($sp)  # save $s3 on stack
            sw $s2, 8($sp)   # save $s2 on stack
            sw $s1, 4($sp)   # save $s1 on stack
            sw $s0, 0($sp)   # save $s0 on stack
            ...  # procedure body
            ...
            exit1: lw $s0, 0($sp) # restore $s0 from stack
            lw $s1, 4($sp)    # restore $s1 from stack
            lw $s2, 8($sp)    # restore $s2 from stack
            lw $s3,12($sp)    # restore $s3 from stack
            lw $ra,16($sp)    # restore $ra from stack
            addi $sp,$sp, 20   # restore stack pointer
            jr $ra             # return to calling routine
```
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

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

### clear2

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

```assembly
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
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
```
Comparison of Array vs. Ptr

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

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Backup
ARM & MIPS Similarities

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

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>$15 \times 32$-bit</td>
<td>$31 \times 32$-bit</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
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
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
    • 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
    • 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPR 0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPR 1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPR 2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPR 3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPR 4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPR 5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPR 6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPR 7</td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
Basic x86 Addressing Modes

- Two operands per instruction

<table>
<thead>
<tr>
<th>Source/dest operand</th>
<th>Second source operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

- Memory addressing modes
  - Address in register
    - Address = $R_{\text{base}} + \text{displacement}$
  - Address = $R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$
### x86 Instruction Encoding

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. JE EIP + displacement</td>
<td>4 4 8</td>
<td>Condition Displacement</td>
</tr>
<tr>
<td>b. CALL</td>
<td>8</td>
<td>Offset</td>
</tr>
<tr>
<td>c. MOV EBX, [EDI + 45]</td>
<td>6 1 1 8 8</td>
<td>d w r/m Postbyte Displacement</td>
</tr>
<tr>
<td>d. PUSH ESI</td>
<td>5 3</td>
<td>Reg</td>
</tr>
<tr>
<td>e. ADD EAX, #6765</td>
<td>4 3 1 32</td>
<td>Reg w Immediate</td>
</tr>
<tr>
<td>f. TEST EDX, #42</td>
<td>7 1 8 32</td>
<td>w Postbyte Immediate</td>
</tr>
</tbody>
</table>
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