Lecture 11:
Sequential processor design

Computer Architecture and
Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012

Last time
- Operators
- Function pointers
- Typedefs and structures
- goto
- Assertions
- Are arrays the same as pointers?
- setjmp() / longjmp()
- Coroutines

Today

- Y86 instruction set architecture
  - Processor state
  - Instruction encoding
  - Compiling, assembling, and simulation
- Sequential processor design
  - Fetch, decode, execute, memory, write back, PC update
  - Performance implications

Why a fictitious ISA?
- Simpler than X86, but close to it
- Illustrate instruction encoding with concrete examples
- Basis for assembler / simulator exercises
- Simple enough to design
  - Sequential processor (this time)
  - Pipelined processor (next that)

Y86 Processor State

Program registers
- Same 8 as with IA32. Each 32 bits
- Condition codes
  - Single-bit flags set by arithmetic or logical instructions
    - OF: Overflow
    - ZF: Zero
    - SF: Negative
- Program counter
  - Indicates address of instruction
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Y86 Instructions

- Format
  - 1-6 bytes of information read from memory
    - Can determine instruction length from first byte
    - Not as many instruction types, and simpler encoding than with IA32
  - Each accesses and modifies some part(s) of the program state
Encoding Registers

• Each register has 4-bit ID
  - Same encoding as in IA32
  - Register ID 8 indicates “no register”
  - Will use this in our hardware design in multiple places

Addition instruction

- Add value in register rA to that in register rB
- Store result in register rB
- Note that Y86 only allows addition to be applied to register data
- Set condition codes based on result
  - e.g., `addl %eax,%esi`

Encoding: 60 06
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

Arithmetic and Logical Operations

• Refer to generically as “OPl”
• Encodings differ only by “function code”
  - Low-order 4 bytes in first instruction word
• Set condition codes as side effect

Move Operations

- Like the IA32 `movl` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Jump Instructions

• Refer to generically as “jXX”
• Encodings differ only by “function code”
• Based on values of condition codes
• Same as IA32 counterparts
• Encode full destination address
  - Unlike PC-relative addressing seen in IA32
Y86 Program Stack

- Region of memory holding program data
- Used in Y86 (and IA32) for supporting procedure calls
- Stack top indicated by %esp
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - When popping, increment stack pointer

Stack Operations

- Decrement %esp by 4
- Store word from rA to memory at %esp
  - Like IA32
- Read word from memory at %esp
- Save in rA
- Increment %esp by 4
  - Like IA32

Subroutine Call and Return

- Push address of next instruction onto stack
- Start executing instructions at Dest
  - Like IA32
- Pop value from stack
- Use as address for next instruction
  - Like IA32

Miscellaneous Instructions

- Don’t do anything
- Stop executing instructions
  - IA32 has comparable instruction, but can’t execute it in user mode
  - We will use it to stop the simulator

Writing Y86 Code

- Try to use C compiler as much as possible
  - Write code in C
  - Compile for IA32 with gcc -S
  - Transliterate into Y86
- Coding example
  - Find number of elements in null-terminated list
    int len1(int a[]);
    int len; for (len = 0; a[len]; len++) ;
    return len;

Y86 code generation example

- First try
  - Write typical array code

  ```c
  /* Find number of elements in null-terminated list */
  int len1(int a[])
  {
    int len;
    for (len = 0; a[len]; len++) ;
    return len;
  }
  ```

- Problem
  - Hard to do array indexing on Y86
    - Since don’t have scaled addressing modes

  ```
  L18: incl %eax
       cmpl $0, (%edx, %eax, 4)
       jne L18
  ```

- Compile with gcc -O2 -S
Y86 code generation example #2

- Second try
  - Write with pointer code

```c
/* Find number of elements in null-terminated list */
int len2(int a[])
{
    int len = 0;
    while (*a++)
        len++;
    return len;
}
```

- Compile with gcc -O2 -S

Y86 code generation example #3

- IA32 code
  - Setup

```y86
len2:
pushl %ebp # Save %ebp
xorl %ecx,%ecx # len = 0
movl %esp,%ebp # Set frame
movl 8(%ebp),%eax # Get a
movl (%edx),%eax # Get *a
jmp L26 # Goto entry
```

- Y86 code
  - Setup

```y86
len2:
pushl %ebp # Save %ebp
xorl %ecx,%ecx # len = 0
movl %esp,%ebp # Set frame
movl 8(%ebp),%eax # Get a
movl (%edx),%eax # Get *a
jmp L26 # Goto entry
```

Y86 code generation example #4

- IA32 code
  - Loop + finish

```y86
L24:
movl (%edx),%eax
incl %ecx
L26:
addl $4,%edx
testl %eax,%eax
jne L24
```

- Y86 code
  - Loop + finish

```y86
L24:
movl (%edx),%eax # Get *a
 irmovl $1,%esi # len++
addl %esi,%ecx # len++
andl %eax,%eax # *a == 0?
jne L24 # No--Loop
rrmovl %ebp,%esp # Pop
rrmovl %ecx,%eax # Rtn len
popl %ebp
ret
```

Y86 Program Structure

- Program starts at address 0
- Must set up stack
  - Make sure don’t overwrite code!
- Must initialize data
  - Can use symbolic names

Assembling Y86 Program

```bash
unix> yas eg.ys
```

- Generates “object code” file eg.yo
  - Actually looks like disassembler output

```
0x000: 388400010000 | irmovl Stack,%esp # Set up stack
0x006: 2045         | rrmovl %esp,%ebp # Set up frame
0x008: 308218000000 | irmovl List,%edx
0x00e: a028         | pushl %edx # Push argument
0x010: 8028000000   | call len2 # Call Function
0x015: 10           | halt # Halt
0x018:              | .align 4 # List of elements
0x018: b3130000     | .long 5043
0x01c: ed170000     | .long 6125
0x020: e31c0000     | .long 7395
0x024: 00000000     | .long 0
```

Simulating Y86 Program

```bash
unix> yis eg.yo
```

- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

```
Stopped in 41 steps at PC = 0x16. Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
    %eax: 0x00000000 0x00000003
    %ecx: 0x00000000 0x00000003
    %edx: 0x00000000 0x00000002
    %esp: 0x00000000 0x00000000
    %ebp: 0x00000000 0x00000000
    %esi: 0x00000000 0x00000000
Changes to memory:
    0x00ff: 0x00000000 0x00000000
    0x00ff: 0x00000000 0x00000000
```
Summary

- Y86 instruction set architecture (ISA)
  - Similar state and instructions as IA32
  - Simpler encodings
  - Somewhere between CISC and RISC

- How important is ISA design?
  - Less now than before
    - With enough hardware, can make almost anything go fast
    - Added complexity of little-used instructions is not many transistors
  - Internally, IA32 is almost RISC anyway
    - Decode unit can generate stream of "RISC-like" instructions

Building Blocks

- Combinational logic
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control

- Storage elements
  - Store bits
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises

HCL Operations

- Classify by type of value returned

  - Boolean expressions
    - Logic operations
      - a && b, a || b, !a
    - Word comparisons
      - \( A \) == \( B \), \( A \) < \( B \), \( A \) <= \( B \), \( A \) > \( B \), \( A \) >= \( B \)
    - Set membership
      - \( A \) in \{ B, C, D \}
        - Same as \( A \) == \( B \) || \( A \) == \( C \) || \( A \) == \( D \)
  - Word expressions
    - Case expressions
      - \{ a : A; b : B; c : C \}
        - Evaluate test expressions \( a, b, c \) ... in sequence
        - Return word expression \( A, B, C \) ... for first successful test

Y86 Instruction Set

- Byte
  - nop
  - halt
  - rmovl V,rB
  - irmovl V,rB
  - rmovl rA,D(rB)
  - mmovl D(rB),rA
  - op1 rA,rB
  - jXX Dest
  - call Dest
  - ret
  - pushl rA
  - popl rA

Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify

  - Data types
    - bool: Boolean
      - \( a, b, c, ... \)
    - int: words
      - \( A, B, C, ... \)
  - Does not specify word size—bytes, 32-bit words, ...

  - Statements
    - bool a = bool-expr ;
    - int A = int-expr ;

SEQ Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
- Memories
  - Access same memory space
  - Data: for reading/writing program data
  - Instruction: for reading instructions

- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter
SEQ Stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter

Instruction Decoding

- Instruction Format
  - Instruction byte \( \text{icode:ifun} \)
  - Optional register byte \( rA:rB \)
  - Optional constant word \( \text{valC} \)

Executing Arith./Logical Operation

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

Stage Computation: Arith/Log. Ops

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing rmmovl

- Fetch
  - Read 6 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
  - Do nothing
- PC Update
  - Increment PC by 6

Stage Computation: rmmovl

- Use ALU for address computation
### Executing popl

- **Fetch**
  - Read 2 bytes
- **Decode**
  - Read stack pointer
- **Execute**
  - Increment stack pointer by 4
- **Memory**
  - Read from old stack pointer
- **Write back**
  - Write result to register
- **PC Update**
  - Increment PC by 2

---

### Stage Computation: popl

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>code:ifun ← M1[PC]</td>
</tr>
<tr>
<td></td>
<td>rA:rB ← M1[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+2</td>
</tr>
<tr>
<td>Decode</td>
<td>valA ← R [⁰ %esp]</td>
</tr>
<tr>
<td></td>
<td>valB ← R [⁰ %esp]</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + 4</td>
</tr>
<tr>
<td>Memory</td>
<td>valM ← M1[valA]</td>
</tr>
<tr>
<td>Write back</td>
<td>R⁰ [⁰ %esp] ← valM</td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valP</td>
</tr>
</tbody>
</table>

- Use ALU to increment stack pointer
- Must update two registers: Popped value, New SP

---

### Executing Jumps

- **Fetch**
  - Read 5 bytes
  - Increment PC by 5
- **Decode**
  - Do nothing
- **Execute**
  - Determine whether to take branch based on jump condition and condition codes

### Stage Computation: Jumps

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>code:ifun ← M1[PC]</td>
</tr>
<tr>
<td></td>
<td>valC ← M1[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+5</td>
</tr>
<tr>
<td>Decode</td>
<td>Bch ← Cond(CC,ifun)</td>
</tr>
<tr>
<td>Execute</td>
<td>take branch?</td>
</tr>
<tr>
<td>Memory</td>
<td>R[⁰ %esp] ← valE</td>
</tr>
<tr>
<td>Write back</td>
<td>R[⁰ %esp] ← valM</td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← Bch ? valC : valP</td>
</tr>
</tbody>
</table>

- Compute both addresses
- Choose based on setting of condition codes and branch condition

---

### Executing call

- **Fetch**
  - Read 5 bytes
  - Increment PC by 5
- **Decode**
  - Read stack pointer
- **Execute**
  - Decrement stack pointer by 4
- **Memory**
  - Write incremented PC to new value of stack pointer
- **Write back**
  - Update stack pointer
- **PC Update**
  - Set PC to Dest

### Stage Computation: call

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>code:ifun ← M1[PC]</td>
</tr>
<tr>
<td></td>
<td>valC ← M1[PC+1]</td>
</tr>
<tr>
<td></td>
<td>valP ← PC+5</td>
</tr>
<tr>
<td>Decode</td>
<td>valB ← R [⁰ %esp]</td>
</tr>
<tr>
<td>Execute</td>
<td>valE ← valB + –4</td>
</tr>
<tr>
<td>Memory</td>
<td>M⁰ [valE] ← valP</td>
</tr>
<tr>
<td>Write back</td>
<td>R⁰ [⁰ %esp] ← valE</td>
</tr>
<tr>
<td>PC update</td>
<td>PC ← valC</td>
</tr>
</tbody>
</table>

- Use ALU to decrement stack pointer
- Store incremented PC
**Executing `ret`**

- **Fetch**
  - Read 1 byte
- **Decode**
  - Read stack pointer
- **Execute**
  - Increment stack pointer by 4

- **Memory**
  - Read return address from old stack pointer
- **Write back**
  - Update stack pointer
- **PC Update**
  - Set PC to return address

---

**Stage Computation: `ret`**

- **Fetch**
  - Read instruction byte
- **Decode**
  - Read operand stack pointer
  - Read operand A
  - Read operand B
  - Compute next PC
- **Execute**
  - Perform ALU operation
  - Set condition code register
  - Read operand B
- **Memory**
  - Read return address from memory
  - Write back memory result
- **Write back**
  - Write back ALU result
  - Update stack pointer
  - Set PC to return address

---

**Computation Steps**

- **Fetch**
  - Read instruction byte
  - Read register byte
  - Read constant word
- **Decode**
  - Compute next PC
  - Read operand A
  - Read operand B
- **Execute**
  - Perform ALU operation
  - Read operand B
- **Memory**
  - Read return address from memory
- **Write back**
  - Write back ALU result
- **PC update**
  - Update PC

---

**Computed Values**

- **Fetch**
  - instruction code
  - instruction function
  - register ID A
- **Decode**
  - register ID B
  - destination register E
- **Execute**
  - ALU result
  - Bch
  - Memory
  - value from memory

---

**SEQ Hardware**

- **Key**
  - **Blue boxes:** predesigned hardware blocks
    - E.g., memories, ALU
  - **Gray boxes:** control logic
    - Describe in HCL
  - **White ovals:** labels for signals
  - **Thick lines:** 32-bit word values
  - **Thin lines:** 4-8 bit values
  - **Dotted lines:** 1-bit values
Fetch Logic

- Predefined Blocks
  - PC: Register containing PC
  - Instruction memory: Read 6 bytes (PC to PC+5)
  - Split: Divide instruction byte into icode and ifun
  - Align: Get fields for rA, rB, and valC

Fetch Control Logic

bool need_regids = icode in { IRRMOVL, IOPL, IPUSHL, IPOPL, IIRMOVL, IRMMOVL, IMRMOVL };
bool instr_valid = icode in { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL, IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

Decode Logic

- Register File
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 8 (no access)
- Control Logic
  - srcA, srcB: read port addresses
  - dstA, dstB: write port addresses

A Source

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Decode</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>rA, rB</td>
<td>Read operand A</td>
<td>None</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>Read operand A</td>
<td>None</td>
</tr>
<tr>
<td>popl rA</td>
<td>Read stack pointer</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>No operand</td>
<td>None</td>
</tr>
<tr>
<td>call Dest</td>
<td>No operand</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>ret</td>
<td>Read stack pointer</td>
<td>Update stack pointer</td>
</tr>
</tbody>
</table>

int srcA = {
  icode in { IRRMOVL, IRMOVL, IOPL, IPUSHL } : rA;
  icode in { IOPL, IRET } : RESP;
  1 : RNONE; # Don't need register
};

E Destination

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Write back</th>
</tr>
</thead>
<tbody>
<tr>
<td>rA, rB</td>
<td>None</td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>None</td>
</tr>
<tr>
<td>popl rA</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>None</td>
</tr>
<tr>
<td>call Dest</td>
<td>Update stack pointer</td>
</tr>
<tr>
<td>ret</td>
<td>Update stack pointer</td>
</tr>
</tbody>
</table>

int dstE = {
  icode in { IRMMOVL, IIRMOVL, IOPL } : rB;
  icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
  1 : RNONE; # Don't need register
};
Execute Logic

- **Units**
  - ALU
    - Implements 4 required functions
    - Generates condition code values
  - CC
    - Register with 3 condition code bits
    - Computes branch flag
- **Control Logic**
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?

\[
\text{ALU A Input}
\]

- Execute
  - \( valE = valB \oplus \text{OP} \) \( valA \)
- \( \text{rrmovl} \ rA, D(rB) \)
- \( \text{popl} \ rA \)
- \( \text{call} \ Dest \)
- \( \text{ret} \)

\[
\text{ALU Operation}
\]

\[
\text{Memory Logic}
\]

- **Memory**
  - Reads or writes memory word
- **Control Logic**
  - Mem. read: should word be read?
  - Mem. write: should word be written?
  - Mem. addr.: Select address
  - Mem. data.: Select data

\[
\text{Memory Address}
\]

\[
\text{Memory Read}
\]

\[
\text{int mem_addr} = \{
\text{icode in} \{ \text{IMRMVOL, IPUSHL, ICALL, IMMRMOVOL} \} : \text{valE} ;
\text{ocode in} \{ \text{IPOPL, IRET} \} : \text{valA} ;
\#\text{ Other instructions don’t need address}
\};
\]

\[
\text{bool mem_read} = \text{icode in} \{ \text{IMRMVOL, IPOPL, IRET} \};
\]
PC Update Logic

- New PC
  - Select next value of PC

SEQ Operation

- State
  - PC register
  - Cond. Code register
  - Data memory
  - Register file
- All updated as clock rises
- Combinational Logic
  - ALU
  - Control logic
  - Memory reads
    - Instruction memory
    - Register file
    - Data memory

SEQ Operation #2

- State set according to second `irmovl` instruction
- Combinational logic starting to react to state changes

SEQ Operation #3

- State set according to second `irmovl` instruction
- Combinational logic generates results for `addl` instruction

SEQ Operation #4

- State set according to `addl` instruction
- Combinational logic starting to react to state changes

int new_pc = {
  icode == ICALL : valC;
  icode == IJXX && Bch : valC;
  icode == IRET : valM;
  1 : valP;
};
SEQ Operation
#5

- state set according to `addl` instruction
- combinational logic generates results for `je` instruction

SEQ Summary

- Implementation
  - Express every instruction as series of simple steps
  - Follow same general flow for each instruction type
  - Assemble registers, memories, predesigned combinational blocks
  - Connect with control logic
- Limitations
  - Too slow to be practical
  - In one cycle, must propagate through instruction memory, register file, ALU, and data memory
  - Would need to run clock very slowly
  - Hardware units only active for fraction of clock cycle