Topics

• Required reading assignment
  – Addressing (Tanenbaum 5.4)
  – Instruction types (Tanenbaum 5.5)

• Reading for your interest
  – Remainder of ISA chapter

• IMUL and Program 3 Grading

• Course evaluations

Addressing Modes

• Immediate
• Direct
• Register
• Register indirect
• Indexed
• Based-indexed
• Stack
Instruction Types

• Dyadic (two operands)
• Monadic (one operand)
• Comparisons and conditional branches
• Procedure calls
• Loop control
• Input/output

Instruction Set Architecture Level

• Interface between software and hardware
• Both compilers and hardware must understand ISA
  – Compilers translate high-level language into object code
  – Hardware must directly execute or interpret ISA
IMUL

- Pops two operands X and Y from the stack, computes the product X * Y, and pushes the product onto the operand stack
  - Opcode: 0x68
  - Multiplying two 32-bit values actually produces a 64-bit result

- Represents an ISA level requirement
  - Could use an IJVM program instead
  - Many methods to implement in microarchitecture

IJVM Implementation

Repeated additions
.method mult(a, b) // OK for small numbers, bad for large ones
.var
product
.end-var
    ILOAD a
START:  BIPUSH 0
    ISTORE product
LOOP:   ILOAD a
        IFEQ END
        ILOAD b
        ILOAD product
        IADD
        ISTORE product
        IINC a -1
        GOTO LOOP
END:    ILOAD product
        IRETURN
.end-method
If Only There Was…

- Hardware functionality for IMUL
  - $TOS = H \times MDR$

- Require numerous AND gates and full adders
  - Typically a separate functional unit with multiple stages

- Web resource

Repeated Additions in Microcode

- Inefficient for large numbers (32-bit)

- Must account for negative numbers
  - Decrement operand if positive
  - Increment operand if negative; then take 2’s complement at the end
Better Implementation

- Use an algorithm based on shifting and adding (with a loop executing 32 times) rather than simple repeated addition
  - Resembles elementary school multiplication

- Potential questions
  - Is looping really necessary?
  - How do you input the counter value?
  - How do you isolate the LSB?
  - Are there enough registers?

One Solution (Slide 1 of 2)

```
imul1  MAR = SP = SP - 1; rd    // Read in next-to-top word of stack
nimul2  OPC = MDR                // Wait for word from memory
nimul3  MDR = CPP; wr            // Copy second stack word to OPC
nimul4  CPP=H=1                 // Save CPP into memory
nimul5  CPP=H=H+CPP             // 1
nimul6  CPP=H=H+CPP             // 10
nimul7  CPP=H=H+CPP             // 100
nimul8  CPP=H=H+CPP             // 1000
nimul9  CPP=H=H+CPP             // 10000
nimul10 MDR = OPC               // Copy second stack word to MDR
nimul11 OPC = 0                 // Set OPC to zero to hold IMUL product
nimul12 H = 1                   // Set H to 1 for bitwise AND
nimul13 Z = TOS AND H; if (Z) goto imul_check1; else goto imul_calculate1
                                 //test to update product
```
One Solution (Slide 2 of 2)

\begin{tabular}{ll}
\textbf{imul\_calculate1} & H = MDR \hspace{1cm} // Copy MDR into H for addition \\
\textbf{imul\_calculate2} & OPC = OPC + H; goto \textbf{imul\_check1} \hspace{1cm} // update OPC with intermediate product \\
\textbf{imul\_check1} & TOS = TOS >> 1 \hspace{1cm} // Shift TOS right by 1 bit \\
\textbf{imul\_check2} & H = MDR \hspace{1cm} // Copy MDR into H for shift left \\
\textbf{imul\_check3} & MDR = H + MDR \hspace{1cm} // Shift MDR left by 1 bit \\
\textbf{imul\_check4} & CPP = CPP - 1; if (Z) goto \textbf{imul\_cleanup1}; else goto \textbf{imul8} \\
& \hspace{1cm} // repeat loop 32 times \\
\textbf{imul\_cleanup1} & TOS = OPC; rd \hspace{1cm} // copy final product into TOS and read in CPP \\
\textbf{imul\_cleanup2} & CPP = MDR \hspace{1cm} // Wait for word from memory \\
\textbf{imul\_cleanup3} & TOS = MDR = TOS \hspace{1cm} // Prepare to write to top of stack \\
\textbf{imul\_cleanup4} & TOS = MDR; wr; goto Main1 \hspace{1cm} // Write to top of stack
\end{tabular}