Machine-Level Programming I: Basics

CENG331 - Computer Organization
Middle East Technical University

Instructors:
Murat Manguoglu (Sections 1-2)

Fall 2019

Slides 3-43 are adapted from the slides of the textbook: D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd Edition
Slides 44-90 are adapted from slides of the textbook: http://csapp.cs.cmu.edu/
Today: Machine Programming I: Basics

- History of Processors, Intel x86 processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
Computer Architecture: A Little History

Why worry about old ideas?

- *Die Geschichte der Wissenschaft ist die Wissenschaft selbst* (The history of science is science itself) – Johann Wolfgang von Goethe, 1749-1832
  
  - In fact, if you read any scientific paper/thesis/work, you will notice it starts with the review of the literature (i.e. History) of the earlier work

- Helps to illustrate the design process, and explains why certain decisions were taken

- Because future technologies might be as constrained as older ones

- **Those who ignore history are doomed to repeat it**
  
  - Every mistake made in mainframe design was also made in minicomputers, then microcomputers, where next?
Antikythera Mechanism
150-100 BC

An astronomical calendar capable of tracking the position of the sun, moon, and planets; predict eclipses, and even recreate the irregular orbit of the moon.

Source: http://en.wikipedia.org/wiki/Antikythera_mechanism
Slide ruler
~1620 and still used today (but only as a backup computer on air...}
Charles Babbage 1791-1871
Lucasian Professor of Mathematics, Cambridge University, 1827-1839
Charles Babbage

- **Difference Engine** 1823
- **Analytic Engine** 1833
  - The forerunner of modern digital computer!

**Application**
- Mathematical Tables – Astronomy
- Nautical Tables – Navy

**Background**
- Any continuous function can be approximated by a polynomial --- *Weierstrass*

**Technology**
- mechanical - gears, Jacquard’s loom, simple calculators
Difference Engine
A machine to compute mathematical tables

Weierstrass:
- Any continuous function can be approximated by a polynomial
- Any polynomial can be computed from difference tables

An example

\[
\begin{align*}
  f(n) &= n^2 + n + 41 \\
  d1(n) &= f(n) - f(n-1) = 2n \\
  d2(n) &= d1(n) - d1(n-1) = 2 \\

  f(n) &= f(n-1) + d1(n) = f(n-1) + (d1(n-1) + 2)
\end{align*}
\]

*all you need is an adder!*

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d2(n)</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>d1(n)</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>f(n)</td>
<td>41</td>
<td>43</td>
<td>47</td>
<td>53</td>
<td>61</td>
</tr>
</tbody>
</table>
Difference Engine

1823
- Babbage’s paper is published

1834
- The paper is read by Scheutz & his son in Sweden

1842
- Babbage gives up the idea of building it; he is onto Analytic Engine!

1855
- Scheutz displays his machine at the Paris World Fare
- Can compute any 6th degree polynomial

*Now the machine is at the Smithsonian*
Analytic Engine

1833: Babbage’s paper was published
- conceived during a hiatus in the development of the difference engine

Inspiration: Jacquard Loom Machine
- looms were controlled by punched cards
  - The set of cards with fixed punched holes dictated the pattern of weave ⇒ program
  - The same set of cards could be used with different colored threads ⇒ numbers

1871: Babbage dies
- The machine remains unrealized.

*It is not clear if the analytic engine could be built using the mechanical technology of the time.*
Analytic Engine

The first conception of a general-purpose computer

1. The store in which all variables to be operated upon, as well as all those quantities which have arisen from the results of the operations are placed.
2. The mill (arithmetic logic unit) into which the quantities about to be operated upon are always brought.

The program

<table>
<thead>
<tr>
<th>Operation</th>
<th>variable1</th>
<th>variable2</th>
<th>variable3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

An operation in the mill required feeding two punched cards and producing a new punched card for the store.

An operation to alter the sequence was also provided!
The first programmer
Ada Byron aka “Lady Lovelace” 1815-52

Ada’s tutor was Babbage himself!
Babbage’s Influence

- Babbage’s ideas had great influence later primarily because of:
  - Luigi Menabrea, who published notes of Babbage’s lectures in Italy
  - Lady Lovelace, who translated Menabrea’s notes in English and thoroughly expanded them.
    “...Analytic Engine weaves algebraic patterns....”

- In the early twentieth century - the focus shifted to analog computers but
  - Harvard Mark I built in 1944 is very close in spirit to the Analytic Engine.
1960 Newmark analogue computer, made up of five units. This computer was used to solve differential equations

ELWAT, Poland, 1967

AKAT-1, Poland, 1959
Linear Equation Solver
John Atanasoff, Iowa State University

1930’s:
- Atanasoff built the Linear Equation Solver.
- It had 300 tubes!
- Special-purpose binary digital calculator
- Dynamic RAM (stored values on refreshed capacitors)

Application:
- Linear and Integral differential equations

Background:
- Vannevar Bush’s Differential Analyzer
  --- an analog computer

Technology:
- Tubes and Electromechanical relays

Atanasoff decided that the correct mode of computation was using electronic binary digits.
Harvard Mark I

- Built in 1944 in IBM Endicott laboratories
  - Howard Aiken – Professor of Physics at Harvard
  - Essentially mechanical but had some electro-magnetically controlled relays and gears
  - Weighed 5 tons and had 750,000 components
  - A synchronizing clock that beat every 0.015 seconds (66Hz)

**Performance:**
- 0.3 seconds for addition
- 6 seconds for multiplication
- 1 minute for a sine calculation
- Decimal arithmetic
- No Conditional Branch!

Broke down once a week!
Electronic Numerical Integrator and Computer (ENIAC)

- Inspired by Atanasoff and Berry, Eckert and Mauchly designed and built ENIAC (1943-45) at the University of Pennsylvania
- The first, completely electronic, operational, general-purpose analytical calculator!
  - 30 tons, 72 square meters, 200KW
- Performance
  - Read in 120 cards per minute
  - Addition took 200 μs, Division 6 ms
  - 1000 times faster than Mark I
- Not very reliable!

Application: Ballistic calculations

angle = f (location, tail wind, cross wind, air density, temperature, weight of shell, propellant charge, ... )
Electronic Discrete Variable Automatic Computer (EDVAC)

- ENIAC’s programming system was external
  - Sequences of instructions were executed independently of the results of the calculation
  - Human intervention required to take instructions “out of order”

- Eckert, Mauchly, John von Neumann and others designed EDVAC (1944) to solve this problem
  - Solution was the stored program computer

  ⇒ “program can be manipulated as data”

- First Draft of a report on EDVAC was published in 1945, but just had von Neumann’s signature!
  - In 1973 the court of Minneapolis attributed the honor of *inventing the computer* to John Atanasoff
Stored Program Computer

**Program = A sequence of instructions**

*How to control instruction sequencing?*

**manual control**

**automatic control**

*external (paper tape)*

Harvard Mark I, 1944

Zuse’s Z1, WW2

*internal*

plug board

ENIAC 1946

read-only memory

ENIAC 1948

read-write memory

EDVAC 1947 *(concept)*

– The same storage can be used to store program and data

**EDSAC 1950 Maurice Wilkes**
## Technology Issues

<table>
<thead>
<tr>
<th>ENIAC</th>
<th>EDVAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>18,000 tubes</td>
<td>4,000 tubes</td>
</tr>
<tr>
<td>20 10-digit numbers</td>
<td>2000 word storage</td>
</tr>
<tr>
<td></td>
<td>mercury delay lines</td>
</tr>
</tbody>
</table>

*ENIAC had many asynchronous parallel units but only one was active at a time*

BINAC: Two processors that checked each other for reliability.

*Didn’t work well because processors never agreed*
Dominant Problem:
Reliability
Mean time between failures (MTBF)
MIT's Whirlwind with an MTBF of 20 min. was perhaps the most reliable machine!
Reasons for unreliability:
1. Vacuum Tubes
2. Storage medium
   - acoustic delay lines
   - mercury delay lines
   - Williams tubes
Selections
Reliability solved by invention of Core memory by J. Forrester 1954 at MIT for Whirlwind project
http://www.wired.com/2010/05/0511magnetic-core-memory/

manguoglu@desktop:~$ some_program_with_a_bug
Segmentation fault (core dumped)
Commercial Activity: 1948-52

IBM’s SSEC (follow on from Harvard Mark I)

*Selective Sequence Electronic Calculator*

- 150 word store.
- Instructions, constraints, and tables of data were read from paper tapes.
- 66 Tape reading stations!
- Tapes could be glued together to form a loop!
- Data could be output in one phase of computation and read in the next phase of computation.
And then there was IBM 701

IBM 701 -- 30 machines were sold in 1953-54 used CRTs as main memory, 72 tubes of 32x32b each

IBM 650 -- a cheaper, drum based machine, more than 120 were sold in 1954 and there were orders for 750 more! *Users stopped building their own machines.*

Why was IBM late getting into computer technology?

*IBM was making too much money!*

Even without computers, IBM revenues were doubling every 4 to 5 years in 40’s and 50’s.
Computers in mid 50’s

- Hardware was expensive
- Stores were small (1000 words) ⇒ No resident system software!
- Memory access time was 10 to 50 times slower than the processor cycle ⇒ Instruction execution time was totally dominated by the memory reference time.
- The ability to design complex control circuits to execute an instruction was the central design concern as opposed to the speed of decoding or an ALU operation
- Programmer’s view of the machine was inseparable from the actual hardware implementation
The IBM 650 (1953-4)

Magnetic Drum (1,000 or 2,000 10-digit decimal words)

Active instruction (including next program counter)

Digit-serial ALU

[From 650 Manual, © IBM]
Programmer’s view of the IBM 650
A drum machine with 44 instructions

Instruction: 60 1234 1009
- “Load the contents of location 1234 into the distribution; put it also into the upper accumulator; set lower accumulator to zero; and then go to location 1009 for the next instruction.”

Good programmers optimized the placement of instructions on the drum to reduce latency!
The Earliest Instruction Sets

*Single Accumulator* - A carry-over from the calculators.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>x</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>x</td>
<td>( AC \leftarrow M[x] )</td>
</tr>
<tr>
<td>STORE</td>
<td>x</td>
<td>( M[x] \leftarrow (AC) )</td>
</tr>
<tr>
<td>ADD</td>
<td>x</td>
<td>( AC \leftarrow (AC) + M[x] )</td>
</tr>
<tr>
<td>SUB</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>x</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td></td>
<td>( AC \leftarrow 2 \times (AC) )</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>x</td>
<td>( PC \leftarrow x )</td>
</tr>
<tr>
<td>JGE</td>
<td>x</td>
<td>if ( (AC) \geq 0 ) then ( PC \leftarrow x )</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>x</td>
<td>( AC \leftarrow \text{Extract address field}(M[x]) )</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Typically less than 2 dozen instructions!
Index Registers

Tom Kilburn, Manchester University, mid 50’s

**One or more specialized registers to simplify address calculation**

Modify existing instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Register</th>
<th>Destination Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>x, IX</td>
<td>AC</td>
<td>(AC \leftarrow M[x + (IX)])</td>
</tr>
<tr>
<td>ADD</td>
<td>x, IX</td>
<td>AC</td>
<td>(AC \leftarrow (AC) + M[x + (IX)])</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add new instructions to manipulate index registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZi</td>
<td>x, IX</td>
<td>if (IX)=0 then PC (\leftarrow x) else IX (\leftarrow (IX) + 1)</td>
</tr>
<tr>
<td>LOADi</td>
<td>x, IX</td>
<td>IX (\leftarrow M[x]) (truncated to fit IX)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Index registers have accumulator-like characteristics
Operations on Index Registers

To increment index register by k

\[
\begin{align*}
    & AC \leftarrow (IX) \quad \text{new instruction} \\
    & AC \leftarrow (AC) + k \\
    & IX \leftarrow (AC) \quad \text{new instruction}
\end{align*}
\]

also the AC must be saved and restored.

It may be better to increment IX directly

\[
\text{INCi} \quad k, \ IX \quad IX \leftarrow (IX) + k
\]

More instructions to manipulate index register

\[
\text{STOREi} \quad x, \ IX \quad M[x] \leftarrow (IX)
\]
(extended to fit a word)

\[
\ldots
\]

IX begins to look like an accumulator

⇒ several index registers
⇒ several accumulators
⇒ General Purpose Registers
Evolution of Addressing Modes

1. Single accumulator, absolute address
   LOAD x

2. Single accumulator, index registers
   LOAD x, IX

3. Indirection
   LOAD (x)

4. Multiple accumulators, index registers, indirection
   LOAD R, IX, x
   or LOAD R, IX, (x) the meaning?
   \[ R \leftarrow M[M[x] + (IX)] \]
   or \[ R \leftarrow M[M[x + (IX)]] \]

5. Indirect through registers
   LOAD Rᵢ, (Rⱼ)

6. The works
   LOAD Rᵢ, Rⱼ, (Rₖ) \[ Rⱼ = \text{index}, Rₖ = \text{base addr} \]
Variety of Instruction Formats

- **One address formats**: Accumulator machines
  - Accumulator is always other source and destination operand

- **Two address formats**: the destination is same as one of the operand sources

  \[(\text{Reg} \times \text{Reg}) \text{ to Reg} \quad R_i \leftarrow (R_i) + (R_j)\]

  \[(\text{Reg} \times \text{Mem}) \text{ to Reg} \quad R_i \leftarrow (R_i) + M[x]\]

  - $x$ can be specified directly or via a register
  - Effective address calculation for $x$ could include indexing, indirection, ...

- **Three address formats**: One destination and up to two operand sources per instruction

  \[(\text{Reg x Reg}) \text{ to Reg} \quad R_i \leftarrow (R_j) + (R_k)\]

  \[(\text{Reg x Mem}) \text{ to Reg} \quad R_i \leftarrow (R_j) + M[x]\]
Zero Address Formats

Operands on a stack

- add: \( M[sp-1] \leftarrow M[sp] + M[sp-1] \)
- load: \( M[sp] \leftarrow M[M[sp]] \)

- Stack can be in registers or in memory (usually top of stack cached in registers)
Burrough’s B5000 Stack Architecture:
An ALGOL Machine, Robert Barton, 1960 -- Dijkstra

- Machine implementation can be completely hidden if the programmer is provided only a high-level language interface.

- **Stack machine organization** because stacks are convenient for:
  1. expression evaluation;
  2. subroutine calls, recursion, nested interrupts;
  3. accessing variables in block-structured languages.

- B6700, a later model, had many more innovative features
  - tagged data
  - virtual memory
  - multiple processors and memories
Hardware organization of the stack

- **Stack is part of the processor state**
  - \[
  \Rightarrow \text{stack must be bounded and small} \\
  \approx \text{number of Registers, not the size of main memory}
  \]

- **Conceptually stack is unbounded**
  - \[
  \Rightarrow \text{a part of the stack is included in the processor state; the rest is kept in the main memory}
  \]
Stack Operations and Implicit Memory References

- Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

  Each push operation ⇒ 1 memory reference
  pop operation ⇒ 1 memory reference

  *No Good!*

- Better performance by keeping the top N elements in registers, and memory references are made only when register stack overflows or underflows.

  *Issue - when to Load/Unload registers?*
Stack Machines (Mostly) Died by 1980

1. Stack programs are not smaller if short (Register) addresses are permitted.

2. Modern compilers can manage fast register space better than the stack discipline.

   GPR’s and caches are better than stack and displays

Early language-directed architectures often did not take into account the role of compilers!

B5000, B6700, HP 3000, ICL 2900, Symbolics 3600

Some would claim that an echo of this mistake is visible in the SPARC architecture register windows
Stacks post-1980

  - Designed to support many parallel processes in Occam language
  - Fixed-height stack design simplified implementation
  - Stack trashed on context swap (fast context switches)
  - Inmos T800 was world’s fastest microprocessor in late 80’s

- **Forth machines**
  - Direct support for Forth execution in small embedded real-time environments
  - Several manufacturers (Rockwell, Patriot Scientific)

- **Java Virtual Machine**
  - Designed for software emulation, not direct hardware execution
  - Sun PicoJava implementation + others

- **Intel x87 floating-point unit**
  - Severely broken stack model for FP arithmetic
  - Deprecated in Pentium-4, replaced with SSE2 FP registers
Software Developments

Machines required experienced operators
⇒ Most users could not be expected to understand these programs, much less write them
⇒ Machines had to be sold with a lot of resident software

Libraries of numerical routines
- Floating point operations
- Transcendental functions
- Matrix manipulation, equation solvers, . . .

1955

High level Languages - Fortran 1956
Operating Systems -
- Assemblers, Loaders, Linkers, Compilers
- Accounting programs to keep track of usage and charges
Compatibility Problem at IBM

By early 60’s, **IBM had 4 incompatible lines of computers!**

<table>
<thead>
<tr>
<th>Old Model</th>
<th>New Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>701</td>
<td>7094</td>
</tr>
<tr>
<td>650</td>
<td>7074</td>
</tr>
<tr>
<td>702</td>
<td>7080</td>
</tr>
<tr>
<td>1401</td>
<td>7010</td>
</tr>
</tbody>
</table>

Each system had its own

- Instruction set
- I/O system and Secondary Storage: magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche
  - business, scientific, real time, ...

⇒ **IBM 360**
IBM 360: Design Premises

Amdahl, Blaauw and Brooks, 1964

- The design must lend itself to *growth and successor machines*
- General method for connecting I/O devices
- Total performance - answers per month rather than bits per microsecond ⇒ *programming aids*
- Machine must be capable of *supervising itself* without manual intervention
- Built-in *hardware fault checking* and locating aids to reduce down time
- Simple to assemble systems with redundant I/O devices, memories etc. for *fault tolerance*
- Some problems required floating-point larger than 36 bits
IBM 360: A General-Purpose Register (GPR) Machine

**Processor State**
- 16 General-Purpose 32-bit Registers
  - may be used as index and base register
  - Register 0 has some special properties
- 4 Floating Point 64-bit Registers
- A Program Status Word (PSW)
  - PC, Condition codes, Control flags

**A 32-bit machine with 24-bit addresses**
- But no instruction contains a 24-bit address!

**Data Formats**
- 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words

The IBM 360 is why bytes are 8-bits long today!
IBM 360: Initial Implementations

<table>
<thead>
<tr>
<th></th>
<th>Model 30</th>
<th>. . .</th>
<th>Model 70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>8K - 64 KB</td>
<td></td>
<td>256K - 512 KB</td>
</tr>
<tr>
<td>Datapath</td>
<td>8-bit</td>
<td></td>
<td>64-bit</td>
</tr>
<tr>
<td>Circuit Delay</td>
<td>30 nsec/level</td>
<td></td>
<td>5 nsec/level</td>
</tr>
<tr>
<td>Local Store</td>
<td>Main Store</td>
<td></td>
<td>Transistor Registers</td>
</tr>
<tr>
<td>Control Store</td>
<td>Read only 1μsec</td>
<td></td>
<td>Conventional circuits</td>
</tr>
</tbody>
</table>

IBM 360 instruction set architecture (ISA) completely hid the underlying technological differences between various models.

Milestone: The first true ISA designed as portable hardware-software interface!

With minor modifications it still survives today!
IBM 360: 47 years later...
The zSeries z11 Microprocessor

- 5.2 GHz in IBM 45nm PD-SOI CMOS technology
- 1.4 billion transistors in 512 mm²
- 64-bit virtual addressing
  - original S/360 was 24-bit, and S/370 was 31-bit extension
- Quad-core design
- Three-issue out-of-order superscalar pipeline
- Out-of-order memory accesses
- Redundant datapaths
  - every instruction performed in two parallel datapaths and results compared
- 64KB L1 I-cache, 128KB L1 D-cache on-chip
- 1.5MB private L2 unified cache per core, on-chip
- On-Chip 24MB eDRAM L3 cache
- Scales to 96-core multiprocessor with 768MB of shared L4 eDRAM

[ IBM, HotChips, 2010]
Intel x86 Processors

- Dominate laptop/desktop/server market

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC – Examples: IBM Power, Arm, ...)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.
x86 Clones: Advanced Micro Devices (AMD)

Historically
- AMD has followed Intel, but sometimes Intel followed AMD too.
- A little bit slower, a lot cheaper

Then
- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

Recent Years
- Intel got its act together
  - Leads the world in semiconductor technology
- AMD has fallen behind
  - Relies on external semiconductor manufacturer
Intel and AMD’s 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing

- **2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (now called “AMD64”)

- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better, IA64 is actually not that bad, latest one was produced in 2017 (Itanium 9760 – 8 cores 2.66Ghz 32MB Cache)

- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!

- **All but low-end x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode
# x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ First 16-bit Intel processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ 1MB address space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ First 32 bit Intel processor, referred to as IA32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Added “flat addressing”, capable of running Unix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4E</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ First 64-bit Intel x86 processor, referred to as x86-64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ First multi-core Intel processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Four cores</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
x86 Processors, cont.

■ Machine Evolution

- 386  1985  0.3M
- Pentium  1993  3.1M
- Pentium/MMX  1997  4.5M
- Pentium Pro  1995  6.5M
- Pentium III  1999  8.2M
- Pentium 4  2001  42M
- Core 2 Duo  2006  291M
- Core i7  2008  731M
- Core i7 Skylake  2015  1.9B
- AMD epyc  2017  19.2B

■ Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores
2018 Intel – Desktop state-of-the-art

- Core i9-9980XE
  - 18 cores
  - 36 threads
  - 3.0-4.4 Ghz
  - 24.75 MB Cache

Image source: https://hwbot.org/newsflash/5002_video_intel_core_i9_7980xe_die_extraction_with_de8auer_(part_2)
2018 AMD – Server state-of-the-art

- **AMD epyc 7742**
  - 64 cores
  - 128 threads
  - 256 MB Cache
  - 2.25-3.4 GHz

Manycore (vs Multicore) processors


Upto 72 cores

(XeonPhi co-processors are no longer produced)
Most Powerful Computers - which processor?

CHIP TECHNOLOGY

source: www.top500.org
Most Powerful Computers – where?

source: www.top500.org
INSIDE INTEL

“A TRULY FASCINATING READ...the first unauthorized history of this highly secretive company.”
—BARRON’S

Andy Grove
and the Rise of the World’s Most Powerful Chip Company

TIM JACKSON

Our Coverage

- **IA32**
  - The traditional x86

- **x86-64**
  - The standard
  - $gcc$ hello.c
  - $gcc$ -m64 hello.c

- **Presentation**
  - 3rd edition covers x86-64
  - 2nd International Edition covers IA32 + x86-64
  - 2nd edition covers only IA32
  - We will only cover x86-64
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
- **Microarchitecture:** Implementation of the architecture.
  - Examples: cache sizes and core frequency.

- **Code Forms:**
  - **Machine Code:** The byte-level programs that a processor executes
  - **Assembly Code:** A text representation of machine code

- **Example ISAs:**
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
Assembly/Machine Code View

**CPU**
- **PC**: Program counter
  - Address of next instruction
  - Called the instruction pointer register “RIP” (x86-64)
- **Register file**: Heavily used program data
- **Condition codes**: Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

**Memory**
- Byte addressable array
- Code and user data
- Stack to support procedures

**Programmer-Visible State**
- **PC**: Program counter
- **Register file**: Heavily used program data
- **Condition codes**: Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`

```
Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`
```

```
```

```
```

<table>
<thead>
<tr>
<th>Text</th>
<th>C program (p1.c p2.c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiler (gcc -Og -S)</td>
</tr>
<tr>
<td>Text</td>
<td>Asm program (p1.s p2.s)</td>
</tr>
<tr>
<td></td>
<td>Assembler (gcc or as)</td>
</tr>
<tr>
<td>Binary</td>
<td>Object program (p1.o p2.o)</td>
</tr>
<tr>
<td></td>
<td>Linker (gcc or ld)</td>
</tr>
<tr>
<td>Binary</td>
<td>Executable program (p)</td>
</tr>
<tr>
<td></td>
<td>Static libraries (.a)</td>
</tr>
</tbody>
</table>

```
Compiling Into Assembly

C Code (sum.c)

```c
long plus(long x, long y);

void sumstore(long x, long y, long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
    pushq %rbx
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    ret
```

Obtain with command

```
gcc -O1 -S sum.c

 gcc -O1 -Wa,-aslh -c sum.c > sum.s
```

Produces file `sum.s`

**Warning:** Will get very different results on different machines due to different versions of gcc and different compiler settings.
Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- Code: Byte sequences encoding series of instructions

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
Object Code

Code for sumstore

0x0400595:
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0xf2
  0xff
  0xff
  0xff
  0x48
  0x89
  0x03
  0x5b
  0xc3

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

- **Assembler**
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution
**Machine Instruction Example**

- **C Code**
  - Store value \( t \) where designated by \( \text{dest} \)

- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - \( t: \) Register \%rax
    - \( \text{dest}: \) Register \%rbx
    - \(*\text{dest}: \) Memory \( M[\%rbx] \)

- **Object Code**
  - 3-byte instruction
  - Stored at address \( 0x40059e \)
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000400595</td>
<td>&lt;sumstore&gt;:</td>
</tr>
<tr>
<td>400595:</td>
<td>53</td>
</tr>
<tr>
<td>400596:</td>
<td>48 89 d3</td>
</tr>
<tr>
<td>400599:</td>
<td>e8 f2 ff ff ff</td>
</tr>
<tr>
<td>40059e:</td>
<td>48 89 03</td>
</tr>
<tr>
<td>4005a1:</td>
<td>5b</td>
</tr>
<tr>
<td>4005a2:</td>
<td>c3</td>
</tr>
</tbody>
</table>

- **Disassembler**
  - `objdump -d sum`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file
Alternate Disassembly

Object

Disassembled

Dump of assembler code for function sumstore:

Disassemble procedure

Within gdb Debugger

gdb sum
disassemble sumstore

- Disassemble procedure

x/14xb sumstore

- Examine the 14 bytes starting at sumstore
What Can be Disassembled?

Anything that can be interpreted as executable code

Disassembler examines bytes and reconstructs assembly source

% objdump -d WINWORD.EXE

WINWORD.EXE: file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:

Reverse engineering forbidden by Microsoft End User License Agreement
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
## x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
Some History: IA32 Registers

16-bit virtual registers (backwards compatibility)

Origin (mostly obsolete)
- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer
Moving Data

- **Moving Data**
  - `movq Source, Dest`:

- **Operand Types**
  - **Immediate**: Constant integer data
    - Example: `$0x400, $-533`
    - Like C constant, but prefixed with `$`
    - Encoded with 1, 2, or 4 bytes
  - **Register**: One of 16 integer registers
    - Example: `%rax, %r13`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - **Memory**: 8 consecutive bytes of memory at address given by register
    - Simplest example: `( %rax )`
    - Various other “address modes”
### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg</td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Imm</td>
<td>Mem</td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction*
Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

  \[
  \text{movq} \ (\%rcx),\%rax
  \]

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \[
  \text{movq} \ 8(\%rbp),\%rdx
  \]
Example of Simple Addressing Modes

```c
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>xp</td>
</tr>
<tr>
<td>%rsi</td>
<td>yp</td>
</tr>
<tr>
<td>%rax</td>
<td>t0</td>
</tr>
<tr>
<td>%rdx</td>
<td>t1</td>
</tr>
</tbody>
</table>

**swap:**

```c
movq    (%rdi), %rax  # t0 = *xp
movq    (%rsi), %rdx  # t1 = *yp
movq    %rdx, (%rdi)  # *xp = t1
movq    %rax, (%rsi)  # *yp = t0
ret```

**Memory**

- **rdi**
- **rsi**
- **rax**
- **rdx**
Understanding Swap()

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

**swap:**

```assembly
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>456</td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

**swap:**

```
movq    (%rdi), %rax  # t0 = *xp
movq    (%rsi), %rdx  # t1 = *yp
movq    %rdx, (%rdi)  # *xp = t1
movq    %rax, (%rsi)  # *yp = t0
ret     
```
Understanding `Swap()`

### Registers

<table>
<thead>
<tr>
<th>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

### swap:

```assembly
movq    (%rdi), %rax  # t0 = *xp
movq    (%rsi), %rdx  # t1 = *yp
movq    %rdx, (%rdi)  # *xp = t1
movq    %rax, (%rsi)  # *yp = t0
ret
```
Understanding Swap()

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

```plaintext
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```
# Understanding Swap()

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>123</td>
</tr>
</tbody>
</table>

**swap:**

```assembly
movq   (%rdi), %rax  ; t0 = *xp
movq   (%rsi), %rdx  ; t1 = *yp
movq   %rdx, (%rdi)  ; *xp = t1
movq   %rax, (%rsi)  ; *yp = t0
ret```

Address
Simple Memory Addressing Modes

- Normal (R) \( \text{Mem}[\text{Reg}[R]] \)
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

\[
\text{movq} \ (\%\text{rcx}) ,\%\text{rax}
\]

- Displacement \( D(R) \) \( \text{Mem}[\text{Reg}[R]+D] \)
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

\[
\text{movq} \ 8(\%\text{rbp}) ,\%\text{rdx}
\]
Complete Memory Addressing Modes

**Most General Form**

\[ D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+D] \]

- **D:** Constant “displacement” 1, 2, or 4 bytes
- **Rb:** Base register: Any of 16 integer registers
- **Ri:** Index register: Any, except for \( %\text{rsp} \)
- **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

**Special Cases**

- \((Rb,Ri)\) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]
- \(D(Rb,Ri)\) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]
- \((Rb,Ri,S)\) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]
## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80,(%rdx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

%rdx 0xf000
%rcx 0x0100
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
Address Computation Instruction

- **leaq** \textit{Src, Dst}
  - \textit{Src} is address mode expression
  - Set \textit{Dst} to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of \( p = \&x[i]; \)
  - Computing arithmetic expressions of the form \( x + k*y \)
    - \( k = 1, 2, 4, \) or \( 8 \)

- **Example**

```c
long m12(long x) {
    return x*12;
}
```

**Converted to ASM by compiler:**

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax    # return t<<2
```
Some Arithmetic Operations

- **Two Operand Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>Dest = Dest – Src</td>
</tr>
<tr>
<td>imulq</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>salq</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorq</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- **One Operand Instructions**
  
  - **incq**: \( Dest = Dest + 1 \)
  
  - **decq**: \( Dest = Dest - 1 \)
  
  - **negq**: \( Dest = -Dest \)
  
  - **notq**: \( Dest = \sim Dest \)

- **See the book or Intel x86-64 manual for more instructions**
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}

Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once

arith:
  leaq (%rdi,%rsi), %rax
  addq %rdx, %rax
  leaq (%rsi,%rsi,2), %rdx
  salq $4, %rdx
  leaq 4(%rdi,%rdx), %rcx
  imulq %rcx, %rax
  ret
Understanding Arithmetic Expression

Example

```c
long arith(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

### arith:
- `leaq (%rdi,%rsi), %rax` # t1
- `addq %rdx, %rax` # t2
- `leaq (%rsi,%rsi,2), %rdx` # t4
- `salq $4, %rdx` # t5
- `lea 4(%rdi,%rdx), %rcx` # rval
- `imulq %rcx, %rax`
- `ret`

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rdx</td>
<td>t4</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>
Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts

- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms

- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation