Introduction to Computer Architecture

Course number: CS141
Who?
Tarun Soni (toni@cs.ucsd.edu)
TA: Wenjing Rao (wen@cs.ucsd.edu) and Eric Liu (xel@cs)
Where?
CENTR: 119
When?
M.W @ 6-8:50pm
Textbook:
Pattern and Hennessy,
Computer Organization & Design
The hardware software interface, 2nd edition.
Web-page: http://www.cs.ucsd.edu/users/toni/cs141
(lecture, homework questions, other pointers and information)
Office hour:
Tarun: Mon. 4pm-6pm: AP&M 3151
Yuan and Wenjing: TBD, look on the webpage

Today's Agenda

- Administration
- Technology trends
- Computer organization: concept of abstraction
- Instruction Set Architectures: Definition, types, examples
- Instruction formats: operands, addressing modes
- Operations: load, store, arithmetic, logical
- Control instructions: branch, jump, procedures
- Stacks
- Examples: in-line code, procedure, nested procedures
- Other architectures

Schedule-sort of

<table>
<thead>
<tr>
<th>Week</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6/30</td>
</tr>
<tr>
<td>2</td>
<td>7/2</td>
</tr>
<tr>
<td>3</td>
<td>7/7</td>
</tr>
<tr>
<td>4</td>
<td>7/9</td>
</tr>
<tr>
<td>5</td>
<td>7/14</td>
</tr>
<tr>
<td>6</td>
<td>7/16</td>
</tr>
<tr>
<td>7</td>
<td>7/21</td>
</tr>
<tr>
<td>8</td>
<td>7/22</td>
</tr>
<tr>
<td>9</td>
<td>7/26</td>
</tr>
<tr>
<td>10</td>
<td>7/30</td>
</tr>
<tr>
<td>11</td>
<td>7/7</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Grading

- Grade breakdown
  - Mid-term (1.5 hours) 30%
  - Final (3 hours) 40%
  - Pop-Quizzes (2. 45 min each, only 2 high scores count) 30%
  - Class Participation: Extra??
- Can't make exams: tell us early and we will work something out
- Homeworks do not need to be turned in. However, pop-quizzes will be based on hw.
- What is cheating?
  - Studying together in groups is encouraged
  - Work must be your own
  - Common examples of cheating: copying an exam question from other material or other person...
  - Better off to skip question (small fraction of grade.)
- Written/email request for changes to grades
  - Average grade will be a B or B+; set expectations accordingly

Why?

- You may become a practitioner someday?
  - Keeper of Moore's law
  - Architecture concepts are core to other sub-systems
    - Video-processors
    - Security engines
    - Routing/Networking etc.
- Even if you become a software geek?
  - Architecture enables a way of thinking
  - Understanding leads to breadth and better implementation of software

'Computer' of the day

Jacquard loom
late 1700's
for weaving silk

"Program" on punch cards

"Microcode": each hole
lifts a set of threads

"Or gate": thread lifted if any controlling hole punched
Trends: Moore's law

- CPU Speed
- DRAM Speed

Trends: $1000 will buy you...

Trends: Densities

- Memory Logic Density
- Transistor Density

Technology

Other technology trends

- Processor
  - Logic capacity: about 30% per year
  - Clock rate: about 20% per year
- Memory
  - DRAM capacity: about 40% per year (4x every 3 years)
  - Memory speed: about 10% per year
  - Cost per bit: about 25% per year
- Disk
  - Capacity: about 60% per year

SPEC Performance

performance now improves - 50% per year (2x every 1.5 years)
Organization: A Basic Computer

Every computer has 5 basic components

- Control
- Memory
- Input
- Data path
- Output

What is “Computer Architecture”

Computer Architecture =
Instruction Set Architecture + Machine Organization

How you talk to the machine          What the machine looks like

Computer Architecture and Engineering

- Instruction Set Design
- Computer Organization
- Interfaces
- Hardware Components
- Compiler/System View
- Logic Designer’s View

Levels of abstraction?

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

- Control Signal Specification

Instruction Set Architecture

ISA is the agreed-upon interface between all the software that runs on the machine and the hardware that executes it.

- Instruction set
- Hardware
- Software
Example ISAs

- IBM360, VAX etc.
- Digital Alpha (v1, v3) 1992-97
- HP PA-RISC (v1.1, v2.0) 1986-96
- Sun Sparc (v8, v9) 1987-95
- SGI MIPS (MIPS I, II, III, IV, V) 1986-96
- Intel (8086, 80286, 80386, 80486, Pentium, MMX, ...)
- ARM ARM7, 8, StrongARM 1995-

Digital Signal Processors also have an ISA
TMS320, Motorola, OAK etc.

Instruction Set Architecture

“How to talk to computers if you aren’t in Star Trek’’

ISAs

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions

We'll be working with the MIPS instruction set architecture
  – similar to other architectures developed since the 1980's
  – used by NEC, Nintendo, Silicon Graphics, Sony

Design goals: maximize performance and minimize cost, reduce design time

Instruction Set Architecture: What Must be Specified?

- Instruction Format or Encoding
  – how is it decoded?
- Location of operands and result
  – where other than memory?
  – how many explicit operands?
  – how are memory operands located?
  – which can or cannot be in memory?
- Data type and size
- Operations
  – what are supported
- Successor instruction
  – jumps, conditions, branches

Fetch-decode-execute is implicit!

Vocabulary

- superscalar processor – can execute more than one instructions per cycle.
- cycle – smallest unit of time in a processor
- parallelism – the ability to do more than one thing at once
- pipelining – overlapping parts of a large task to increase throughput without decreasing latency
CRAFTING AN ISA

- We'll look at some of the decisions facing an instruction set architect, and
- how those decisions were made in the design of the MIPS instruction set.

MIPS, like SPARC, PowerPC, and Alpha AXP, is a RISC (Reduced Instruction Set Computer) ISA.
- fixed instruction length
- few instruction formats
- load/store architecture
- RISC architectures worked because they enabled pipelining. They continue to thrive because they enable parallelism.

BASIC TYPES OF ISAS

**Accumulator (1 register):**
1 address: `add A, acc ← acc + mem[A]`
1x address: `adds A, acc ← acc + mem(A + x)`

**Stack:**
0 address: `add tos ← tos + next`

**General Purpose Register:**
2 address: `add A, B, EA(A) ← EA(A) + EA(B)`
3 address: `add A, B, C, EA(A) ← EA(B) + EA(C)`

**Load/Store:**
3 address: `add Ra, Rb, Re, Ra ← Rb + Re`
load Ra, Rb: `Ra ← mem[Rb]`
store Ra, Rb: `mem[Rb] ← Ra`

Comparison:
- Bytes per instruction?
- Number of Instructions?
- Cycles per instruction?

INSTRUCTION LENGTH

<table>
<thead>
<tr>
<th>Variable:</th>
<th>Fixed:</th>
<th>Hybrid:</th>
</tr>
</thead>
</table>

MIPS Instructions:
- All instructions have 3 operands
- Operand order is fixed (destination first)

C code: `C = A + B`
MIPS code: `add s0, s1, s2` (associated with variables by compiler)

Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16) choose performance or density per procedure
MIPS Instruction Format

- 6 bits
- 5 bits
- 5 bits
- 5 bits
- 5 bits
- 6 bits

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- OP: operation code that tells the machine which format
- rs, rt, rd, sa: registers
- funct: function code

- Immediate: 0, 1, 2, 3
- Register: 6 useless bits
- Shift: 4 0s
- JAL: 5 shifts

- Operands are generally in one of two places:
  - registers (32 int, 32 fp)
  - memory (2^32 locations)
- Registers are:
  - easy to specify
  - close to the processor (fast access)
- The idea that we want to access registers whenever possible led to:
  - load-store architectures
  - normal arithmetic instructions only access registers
  - only access memory with explicit loads and stores

Operands

- Operands are generally in one of two places:
  - registers (32 int, 32 fp)
  - memory (2^32 locations)
- Registers are:
  - easy to specify
  - close to the processor (fast access)
- The idea that we want to access registers whenever possible led to:
  - load-store architectures
  - normal arithmetic instructions only access registers
  - only access memory with explicit loads and stores

Operands

Expect new instruction set architecture to use

- general purpose register
- Pipelining => Expect it to use load store variant of GPR ISA

Load-Store Architectures

- Can do:
  - add r1=r2+r3
  - load r3, M(address)

- Can't do:
  - add r1 = r2 + M(address)
  - load r3, M(address)

- Forces heavy dependence on
  - registers, which is exactly what
  - you want in today's CPUs

- Expect new instruction set architecture to use

- General Purpose Registers

- Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
    - e.g., (A*B) * (C*D) = (A*E) * (D*F) multiplies in any order vs. stack
    - registers can hold variables
    - memory traffic is reduced, so program is sped up
    - code density improves (since register named with fewer bits
    - than memory location)

- MIPS Registers

- Programmable storage
  - 2^32 x bytes of memory
    - 31 x 32-bit GPRs (R0 = 0)
    - 32 x 32-bit FP regs (paired DF)
  - HI, LO, PC

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.

- Bytes are nice, but most data items use larger "words"

- For MIPS, a word is 32 bits or 4 bytes.

- 2^32 bytes with byte addresses from 0 to 2^32-1
- 2^30 words with byte addresses 0, 4, 8, ..., 2^30-4
- Words are aligned
  - i.e., what are the least 2 significant bits of a word address?
Addressing: Endian-ness and alignment

- Big Endian: address of most significant byte = word address (x00 = Big End of word)
  - IBM 360/370, Motorola 68K, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address (x00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>big endian byte</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>little endian byte</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Alignment: object falls on address that is multiple of their size.</td>
<td>Not Aligned</td>
<td>Aligned</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes: Usage

3 programs measured on machine with all address modes (VAX)

- Displacement: 42% avg, 32% to 55%  75%  85%
- Immediate: 33% avg, 17% to 43%  85%
- Register deferred (indirect): 13% avg, 3% to 24%  85%
- Scaled: 7% avg, 0% to 16%
- Memory indirect: 3% avg, 1% to 6%
- Misc: 2% avg, 0% to 3%

75% displacement & immediate 88% displacement, immediate & register indirect

similar measurements:
- 16 bits is enough for the immediate address 75 to 80% of the time
- 16 bits is enough of a displacement 99% of the time
### Addressing mode usage: Application Specific

<table>
<thead>
<tr>
<th>Program</th>
<th>Base + Displacement</th>
<th>Immediate</th>
<th>Scaled Index</th>
<th>Memory Indirect</th>
<th>All Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEX</td>
<td>56%</td>
<td>43%</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Space</td>
<td>58%</td>
<td>17%</td>
<td>13%</td>
<td>6%</td>
<td>3%</td>
</tr>
<tr>
<td>GCC</td>
<td>51%</td>
<td>39%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
</tr>
</tbody>
</table>

### MIPS Addressing Modes

- **Register direct**: 
  - `OP rs rt rd sa funct`
  - `add $1, $2, $3`

- **Immediate**: 
  - `OP rs rt immediate`
  - `add $1, $2, 35`

- **Base + displacement**: 
  - `be $1, disp(32)`
  - `register indirect` 
  - `rs disp = 0`
  - `(rt) = 0`

### MIPS ISA so far

- fixed 32-bit instructions
- 3 instruction formats
- 3-operator, load-store architecture
- 32 general-purpose registers (integer, floating point)
  - `R0` always equals 0.
- 2 special purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
- registers are 32-bits wide (word)
- register, immediate, and base+displacement addressing modes

But what about the actual instructions themselves ??

### Typical Operations (little change since 1966)

- **Data Movement**:
  - Load (from memory)
  - Store (to memory)
  - memory-to-memory move
  - register-to-register move
  - input (from I/O device)
  - op (op from stack)

- **Arithmetic**:
  - Integer (binary or decimal) or FP
  - Add, Subtract, Multiply, Divide

- **Shift**:
  - shift left/right, rotate left/right

- **Logical**:
  - not, and, or, set, clear

- **Control** (Jump/Branch)
  - unconditional, conditional

- **Subroutine Linkage**:
  - call, return

- **Interrupt**:
  - trap, return

- **Synchronization**:
  - lock & unlock (atomic r-mw)

- **String**:
  - search, translate

- **Graphics (MMX)**:
  - parallel subword ops (4 16-bit add)

### 80x86 Instruction usage

- **Rank instruction**
  - Integer Average Percent total executed
  - 1 load 22%
  - 2 conditional branch 20%
  - 3 compare 16%
  - 4 store 12%
  - 5 add 8%
  - 6 and 6%
  - 7 sub 5%
  - 8 move register-register 4%
  - 9 call 1%
  - 10 return 1%

- **Simple instructions dominate instruction frequency**

### Instruction usage

- Support the simple instructions, since they will dominate the number of instructions executed:
  - load, store, add, subtract, move register-register, and, or, shift, compare equal, compare not equal, branch, jump, call, return;

### Compiler issues

- **Unoptimizing**:
  - no special registers, few special cases, all operand modes available with any data type or instruction type

- **Completeness**:
  - support for a wide range of operating systems, target applications

- **Regularity**:
  - no overlapping for the meanings of instruction fields

- **Unresolved**:
  - resource needs easily determined

- **Register Assignment**:
  - critical too

- Easier if lots of registers
MIPS Instructions

- arithmetic
  - add, subtract, multiply, divide
- logical
  - and, or, shift left, shift right
- data transfer
  - load word, store word
- conditional Branch
  - unconditional Branch

- data transfer
  - load word, store word

MIPS Control Instructions

- How do you specify the destination of a branch/jump?
  - studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else)
  - we can specify a relative address in much fewer bits than an absolute address
    - e.g., beq $1, $2, 100  if ($1 = $2) PC = PC + 100 * 4
- How do we specify the condition of the branch?
  - Condition Codes
    - Processor status bits are set as a side-effect of arithmetic instructions (possibly on Move) or explicitly by compare or test instructions.
    - add $r1, $r2, $r3  bz label
    - cmp $r1, $r2, $r3  bgt label
    - compare and Branch
      - bgt $r1, $r2, label

Conditional Branch Distance

- Instruction Average
- FP Average

Bits of Branch Displacement

- Instruction Average
- FP Average

10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
0% 10 20 30 40 50 60 70 80 90 100
Conditional Branching
- PC-relative since most branches are relatively close to the current PC address.
- At least 8 bits suggested (≤ 128 instructions)
- Compare Equal/Not Equal most important for integer programs (85%)

Jumps
- need to be able to jump to an absolute address sometime
- need to be able to do procedure calls and returns
- jump – j 10000 => PC = 10000
- Jump and link – jal 100000 => $31 = PC + 4; PC = 10000
  - used for procedure calls
- jump register – jr $31 => PC = $31
  - used for returns, but can be useful for lots of other things.

MIPS Branch & Jump Instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1, $2, 100</td>
<td>if ($1 = $2) go to PC+4+100</td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1, $2, 100</td>
<td>if ($1 ≠ $2) go to PC+4+100</td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1, $2, $3</td>
<td>if ($2 &lt; $3) $1 = 1; else $1 = 0</td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1, $2, 100</td>
<td>if ($2 &lt; 100) $1 = 1; else $1 = 0</td>
</tr>
<tr>
<td>set less than unms.</td>
<td>sltiu $1, $2, $3</td>
<td>if ($2 &lt; $3) $1 = 1; else $1 = 0</td>
</tr>
<tr>
<td>set lt. imm. unms.</td>
<td>sll $1, $2, $3</td>
<td>if ($2 &lt; 100) $1 = 1; else $1 = 0</td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000</td>
</tr>
</tbody>
</table>

MIPS Instruction Formats:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>OP</td>
<td>target</td>
<td></td>
</tr>
</tbody>
</table>

MIPS-Addressing Formats: Branches and Jumps
- Branches (e.g., beq) use PC-relative addressing mode (few bits if addr typically close)
- Some have replacement modes, with the PC being the base.
- Jump uses pseudo-direct addressing mode: 26 bits of the address is in the instruction, the rest is taken from the PC.

Stacks

Stacking of Subroutine Calls & Returns and Environments:
- Some machines provide a memory stack as part of the architecture (e.g., VAX)
- Sometimes stacks are implemented via software convention (e.g., MIPS)

Frequency of comparison types in branches

- LT/GE
- 7%
- 40%
- GT/LE
- 23%
- EQ/NE
- 17%
- FP Avg.
- 86%
- Int Avg.
Word from memory to register

if $r$ sequence words differ by 4. Memory holds data structures, such as arrays.

Three operands; data in registers

$\text{ss} = \text{ss} + 100$

$\text{sp} = \text{sp} + 1$

If

$\text{sp}[\text{sp}] = \text{ss}$

Jump to target address

Load from upper 16 bits

$\text{ss} = 1$

If

$\text{ds} < \text{ds}$

$\text{sp} = \text{sp} + 1$

$\text{sp} = \text{sp} + 1$

$\text{sp} = \text{sp} + 1$

Stacks

Useful for stacked environments/subroutine call & return even if

opend stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:

Next Empty?

inf. Big

grows up
grows down

0 Little

Memory Addresses

Little $\rightarrow$ Big/Last Full

Little $\rightarrow$ Big/Next Empty

POP: Read from Mem(SP)
Decrement SP

PUSH: Increment SP
Write to Mem(SP)

MIPS Software Register Conventions

0: zero constant 0
1: at reserved for assembler

2: $v0$ expression evaluation &

3: $v1$ function results

4: $a0$ arguments

5: $a1$

6: $a2$

7: $a3$

8: $t0$ temporary: caller saves
... (callee can clobber)

10: $t1$ temporary: caller saves
... (callee can clobber)

15: $t6$

16: $s0$ callee saves
... (callee can clobber)

17: $s1$

MIPS Branch & Jump Instructions

Example: Swap(i)

Example: Leaf_procedure()

Are we reading the code?

...
**Example: Nested procedure**

```c
int fact(int n) {
    if (n==1) return(1);
    else return (n*fact(n-1));
}
```

- What about nested procedures? Size?
- Recursive procedures?

```c
Resume $a0 = n
    fact
    sub $gp,$gp,0     // Make space for 2 temp locations
    sw $a0,4($gp)     // save return address
    lw $t0,$a0,1     // test for n!
    bge $t0,$a0,3,63  // if (n==1) goto LI
    add $v0,$t0,1     // $v0 = n!
    li $t0,1
    add $sp,$sp,8     // 'pop' the stack
    jr $ra

LI: sub $a0,$a0,1     // n--;
    jal fact;           // call fact again.
    add $sp,$sp,8       // return
}
```

**Other Architectures**

- Design alternative:
  - provide more powerful operations (e.g., DSP, Encryption engines, Java Processors)
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and a higher CPI
- Sometimes referred to as "RISC vs. CISC"
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
  - instructions from 1 to 54 bytes long!
- We'll look at PowerPC and 80x86

**PowerPC**

- Indexed addressing
  - example: lw $t1,$a0+1($s3)
  - What do we have to do in MIPS?
- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: lw $v0,$a0+3($s3)  // $v0 = Memory[$a0+3]
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register "bc" Loop
  - decrement counter, if not 0 goto loop
  - VAX: minimize code size, make assembly language easy
  - danger is a slower cycle time and/or a higher CPI
  - goal is to reduce number of instructions executed
  - provide more powerful operations (e.g., DSP, Encryption engines, Java Processors)

**80x86: Volume is beautiful**

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, adds instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
  (mostly designed for higher performance)
- 1997: MMX is added

“This history illustrates the impact of the "golden handcuffs" of compatibility”

"adding new features as someone might add clothing to a packed bag"

“an architecture that is difficult to explain and impossible to love”

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”

**80x86: Complex Instruction Set**

- See text for a detailed description...
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
  - e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

**Comparing Instruction Set Architectures**

Design-time metrics:
- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

Static Metrics:
- How many bytes does the program occupy in memory?
- Dynamic Metrics:
  - How many instructions are executed?
  - How many bytes does the processor fetch to execute the program?

How many clocks are required per instruction?

How "lean" a clock is practical?

Best Metric: Time to execute the program

This depends on
- instruction set,
- processor organization, and
- compilation techniques.
### Instruction Set Architectures: What did we learn today?

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count.
- Four principles of IS architecture:
  - Simplicity favors regularity
  - Smaller is faster
  - Good design demands compromise
  - Make the common case fast

### Todays Agenda

- Administrivia
- Technology trends
- Computer organization: concept of abstraction
- Instruction Set Architectures: Definition, types, examples
- Instruction formats: operands, addressing modes
- Operations: load, store, arithmetic, logical
- Control instructions: branch, jump, procedures
- Stacks
- Examples: in-line code, procedure, nested-procedures
- Other architectures