

15-740/18-740
Computer Architecture
Lecture 4: ISA Tradeoffs

Prof. Onur Mutlu
Carnegie Mellon University
Fall 2011, 9/19/2011

Review of Last Lecture

- Intro to on-chip networks
- More on DRAM controllers and bank operation
- Begin ISA tradeoffs
 - Semantic gap

Today

- More ISA tradeoffs

Review: Levels of Transformation

■ ISA

- Agreed upon interface between software and hardware
 - SW/compiler assumes, HW promises
- What the software writer needs to know to write system/user programs

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

■ Microarchitecture

- Specific implementation of an ISA
- Not visible to the software

■ Microprocessor

- **ISA, uarch**, circuits
- “Architecture” = ISA + microarchitecture

Review: ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
 - Gas pedal: interface for “acceleration”
 - Internals of the engine: implements “acceleration”
 - Add instruction vs. Adder implementation

- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
 - Bit serial, ripple carry, carry lookahead adders
 - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, ...

- Uarch usually changes faster than ISA
 - Few ISAs (x86, SPARC, MIPS, Alpha) but many uarchs
 - *Why?*

Review: Microarchitecture

- Implementation of the ISA under specific design constraints and goals
- Anything done in hardware without exposure to software
 - Pipelining
 - In-order versus out-of-order instruction execution
 - Memory access scheduling policy
 - Speculative execution
 - Superscalar processing (multiple instruction issue?)
 - Clock gating
 - Caching? Levels, size, associativity, replacement policy
 - Prefetching?
 - Voltage/frequency scaling?
 - Error correction?

Review: Tradeoffs: Soul of Computer Architecture

- ISA-level tradeoffs
- Uarch-level tradeoffs
- System and Task-level tradeoffs
 - How to divide the labor between hardware and software

Review: ISA-level Tradeoffs: Semantic Gap

- Where to place the ISA? Semantic gap
 - Closer to high-level language (HLL) or closer to hardware control signals? → Complex vs. simple instructions
 - RISC vs. CISC vs. HLL machines
 - FFT, QUICKSORT, POLY, FP instructions?
 - VAX INDEX instruction (array access with bounds checking)
 - Tradeoffs:
 - Simple compiler, complex hardware vs. complex compiler, simple hardware
 - Caveat: Translation (indirection) can change the tradeoff!
 - Burden of backward compatibility
 - Performance?
 - Optimization opportunity: Example of VAX INDEX instruction: who (compiler vs. hardware) puts more effort into optimization?
 - Instruction size, code size

Review: X86: Small Semantic Gap: String Operations

REP MOVSB DEST SRC

```
IF AddressSize = 16
  THEN
    Use CX for CountReg;
  ELSE IF AddressSize = 64 and REX.W used
    THEN Use RCX for CountReg; FI;
  ELSE
    Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
  DO
    Service pending interrupts (if any);
    Execute associated string instruction;
    CountReg ← (CountReg - 1);
    IF CountReg = 0
      THEN exit WHILE loop; FI;
    IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
      or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
      THEN exit WHILE loop; FI;
  OD;
```

```
DEST ← SRC;
IF (Byte move)
  THEN IF DF = 0
    THEN
      (R)ESI ← (R)ESI + 1;
      (R)EDI ← (R)EDI + 1;
    ELSE
      (R)ESI ← (R)ESI - 1;
      (R)EDI ← (R)EDI - 1;
    FI;
  ELSE IF (Word move)
    THEN IF DF = 0
      (R)ESI ← (R)ESI + 2;
      (R)EDI ← (R)EDI + 2;
    FI;
    ELSE
      (R)ESI ← (R)ESI - 2;
      (R)EDI ← (R)EDI - 2;
    FI;
  ELSE IF (Doubleword move)
    THEN IF DF = 0
      (R)ESI ← (R)ESI + 4;
      (R)EDI ← (R)EDI + 4;
    FI;
    ELSE
      (R)ESI ← (R)ESI - 4;
      (R)EDI ← (R)EDI - 4;
    FI;
  ELSE IF (Quadword move)
    THEN IF DF = 0
      (R)ESI ← (R)ESI + 8;
      (R)EDI ← (R)EDI + 8;
    FI;
    ELSE
      (R)ESI ← (R)ESI - 8;
      (R)EDI ← (R)EDI - 8;
    FI;
FI;
```

How many instructions does this take in Alpha?

Small Semantic Gap Examples in VAX

- FIND FIRST
 - Find the first set bit in a bit field
 - Helps OS resource allocation operations
- SAVE CONTEXT, LOAD CONTEXT
 - Special context switching instructions
- INSQUEUE, REMQUEUE
 - Operations on doubly linked list
- INDEX
 - Array access with bounds checking
- STRING Operations
 - Compare strings, find substrings, ...
- Cyclic Redundancy Check Instruction
- EDITPC
 - Implements editing functions to display fixed format output
- Digital Equipment Corp., “[VAX11 780 Architecture Handbook](#),” 1977-78.

Small versus Large Semantic Gap

- CISC vs. RISC
 - Complex instruction set computer → complex instructions
 - Initially motivated by “not good enough” code generation
 - Reduced instruction set computer → simple instructions
 - John Cocke, mid 1970s, IBM 801 (see ASPLOS 1982 paper)
 - Goal: enable better compiler control and optimization

- RISC motivated by
 - Memory stalls (no work done in a complex instruction when there is a memory stall?)
 - When is this correct?
 - Simplifying the hardware → lower cost, higher frequency
 - Enabling the compiler to optimize the code better
 - Find fine-grained parallelism to reduce stalls

Small versus Large Semantic Gap

- John Cocke's RISC (large semantic gap) concept:
 - Compiler generates control signals: open microcode
- Advantages of Small Semantic Gap (Complex instructions)
 - + Denser encoding → smaller code size → saves off-chip bandwidth, better cache hit rate (better packing of instructions)
 - + Simpler compiler
- Disadvantages
 - Larger chunks of work → compiler has less opportunity to optimize
 - More complex hardware → translation to control signals and optimization needs to be done by hardware
- Read Colwell et al., “[Instruction Sets and Beyond: Computers, Complexity, and Controversy](#),” IEEE Computer 1985.

ISA-level Tradeoffs: Instruction Length

- **Fixed length:** Length of all instructions the same
 - + Easier to decode single instruction in hardware
 - + Easier to decode multiple instructions concurrently
 - Wasted bits in instructions (Why is this bad?)
 - Harder-to-extend ISA (how to add new instructions?)
- **Variable length:** Length of instructions different (determined by opcode and sub-opcode)
 - + Compact encoding (Why is this good?)
 - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. How?
 - More logic to decode a single instruction
 - Harder to decode multiple instructions concurrently
- **Tradeoffs**
 - Code size (memory space, bandwidth, latency) vs. hardware complexity
 - ISA extensibility and expressiveness
 - Performance? Smaller code vs. imperfect decode

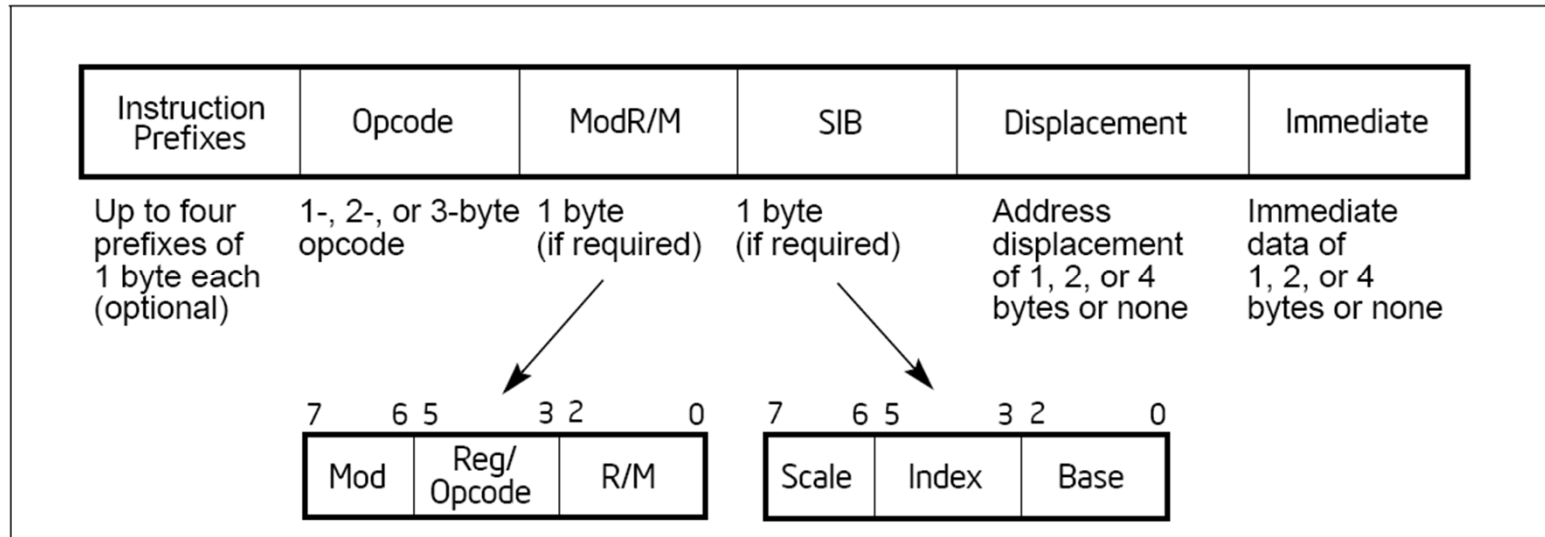
ISA-level Tradeoffs: Uniform Decode

- **Uniform decode:** Same bits in each instruction correspond to the same meaning
 - Opcode is always in the same location
 - Ditto operand specifiers, immediate values, ...
 - Many “RISC” ISAs: Alpha, ARM, MIPS, PowerPC, SPARC
 - + Easier decode, simpler hardware
 - + Enables parallelism: generate target address before knowing the instruction is a branch
 - Restricts instruction format (fewer instructions?) or wastes space

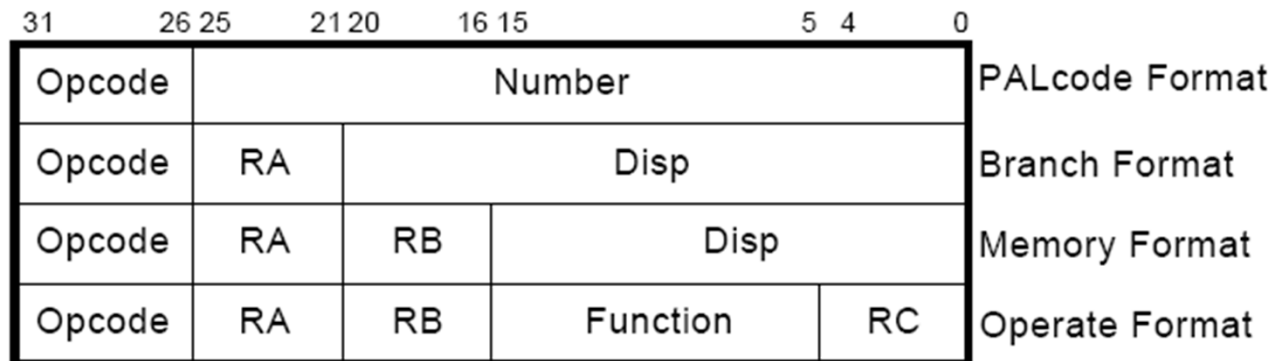
- **Non-uniform decode**
 - E.g., opcode can be the 1st-7th byte in x86
 - + More compact and powerful instruction format
 - More complex decode logic (e.g., more logic to speculatively generate branch target)

x86 vs. Alpha Instruction Formats

■ x86:



■ Alpha:



ISA-level Tradeoffs: Number of Registers

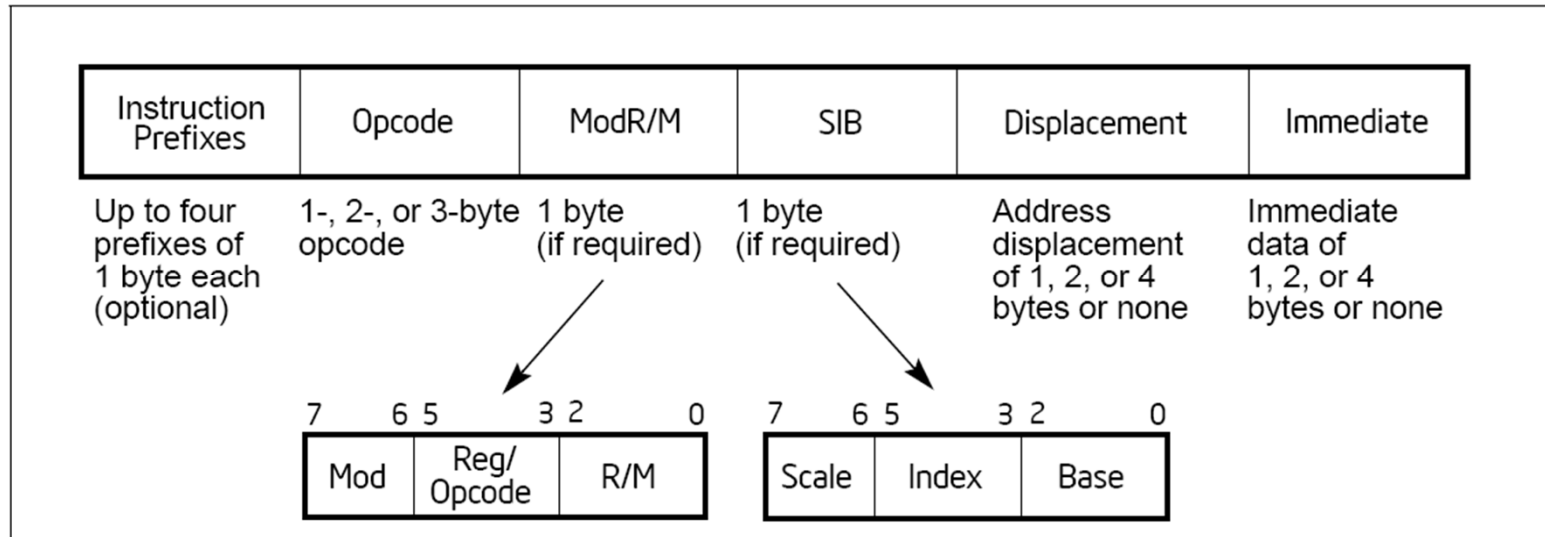
- Affects:
 - Number of bits used for encoding register address
 - Number of values kept in fast storage (register file)
 - (uarch) Size, access time, power consumption of register file
- Large number of registers:
 - + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
 - Larger instruction size
 - Larger register file size
 - (Superscalar processors) More complex dependency check logic

ISA-level Tradeoffs: Addressing Modes

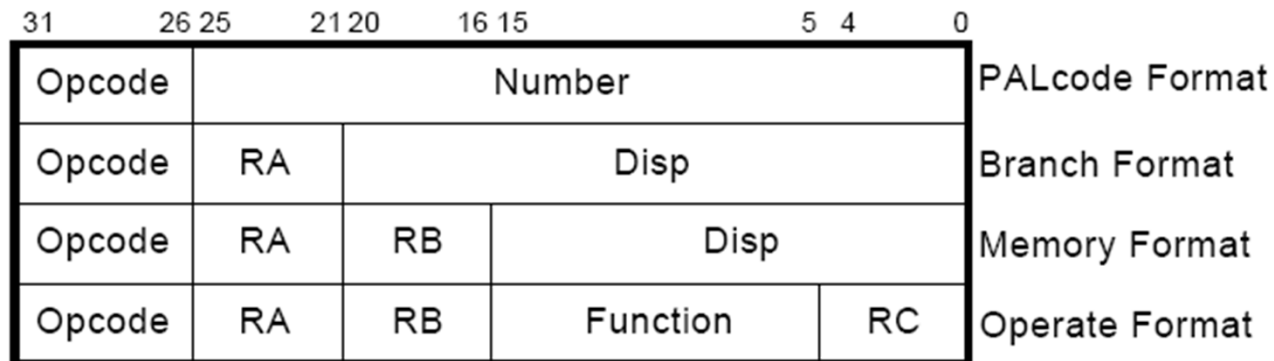
- Addressing mode specifies how to obtain an operand of an instruction
 - Register
 - Immediate
 - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)
- More modes:
 - + help better support programming constructs (arrays, pointer-based accesses)
 - make it harder for the architect to design
 - too many choices for the compiler?
 - Many ways to do the same thing complicates compiler design
 - Read *Wulf*, “*Compilers and Computer Architecture*”

x86 vs. Alpha Instruction Formats

■ x86:



■ Alpha:



x86

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (in decimal) /digit (Opcode) (in binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] [---] ¹ disp32 ² [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8 ³ [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [---]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [---]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

NOTES:

1. The [---] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.

x86

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] none [EBP] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
[EAX*2] [ECX*2] [EDX*2] [EBX*2] none [EBP*2] [ESI*2] [EDI*2]	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
[EAX*4] [ECX*4] [EDX*4] [EBX*4] none [EBP*4] [ESI*4] [EDI*4]	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
[EAX*8] [ECX*8] [EDX*8] [EBX*8] none [EBP*8] [ESI*8] [EDI*8]	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

indexed
(base +
index)

scaled
(base +
index*4)

NOTES:

- The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits Effective Address

00 [scaled index] + disp32

01 [scaled index] + disp8 + [EBP]

10 [scaled index] + disp32 + [EBP]

Other ISA-level Tradeoffs

- Load/store vs. Memory/Memory
- Condition codes vs. condition registers vs. compare&test
- Hardware interlocks vs. software-guaranteed interlocking
- VLIW vs. single instruction vs. SIMD
- 0, 1, 2, 3 address machines (stack, accumulator, 2 or 3-operands)
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Aligned vs. unaligned access
- Supported data types
- Software vs. hardware managed page fault handling
- Granularity of atomicity
- Cache coherence (hardware vs. software)
- ...

Programmer vs. (Micro)architect

- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job
- Virtual memory
 - vs. overlay programming
 - Should the programmer be concerned about the size of code blocks?
- Unaligned memory access
 - Compile/programmer needs to align data
- Transactional memory?
- VLIW vs. SIMD? Superscalar execution vs. SIMD?