Computer Architecture

Lecture 3: ISA Tradeoffs

Dr. Ahmed Sallam

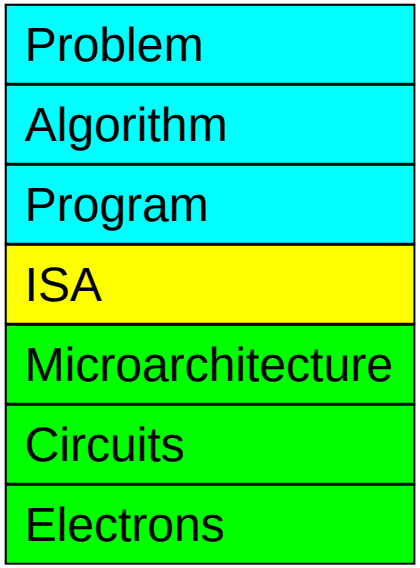
Suez Canal University

Spring 2015

Based on original slides by Prof. Onur Mutlu

Design Point

- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Considerations
  - Cost
  - Performance
  - Maximum power consumption
  - Energy consumption (battery life)
  - Availability
  - Reliability and Correctness
  - Time to Market



- Design point determined by the "Problem" space (application space), or the intended users/*market*

Application Space

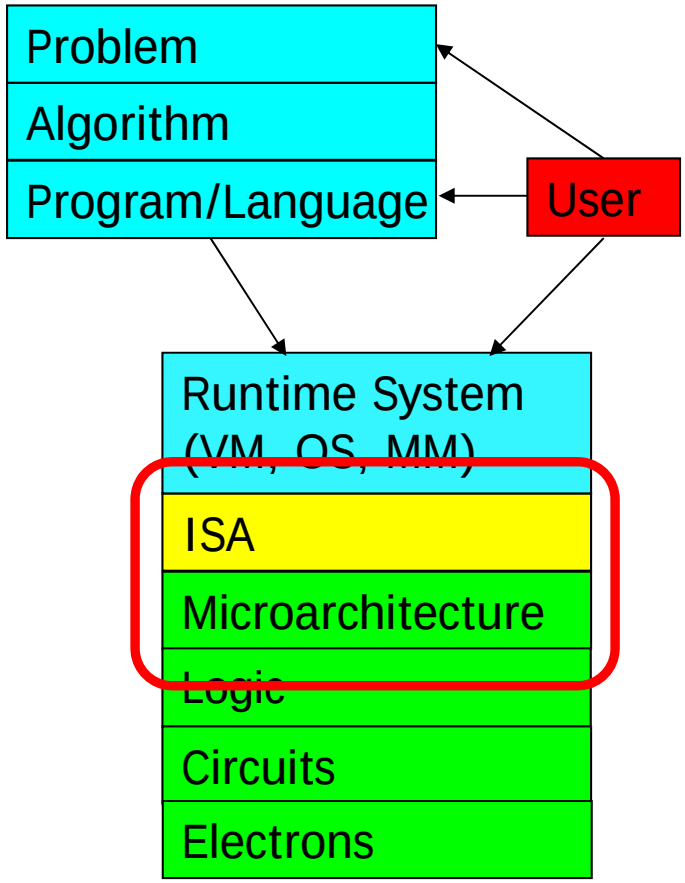
- Dream, and they will appear...

Other examples of the application space that continue to drive the need for unique design points are the following:

1) scientific applications such as those whose computations control nuclear power plants, determine where to drill for oil, and predict the weather;
2) transaction-based applications such as those that handle ATM transfers and e-commerce business;
3) business data processing applications, such as those that handle inventory control, payrolls, IRS activity, and various personnel record keeping, whether the personnel are employees, students, or voters;
4) network applications, such as high-speed routing of Internet packets, that enable the connection of your home system to take advantage of the Internet;
5) guaranteed delivery (a.k.a. real time) applications that require the result of a computation by a certain critical deadline;
6) embedded applications, where the processor is a component of a larger system that is used to solve the (usually) dedicated application;
7) media applications such as those that decode video and audio files;
8) random software packages that desktop users would like to run on their PCs.

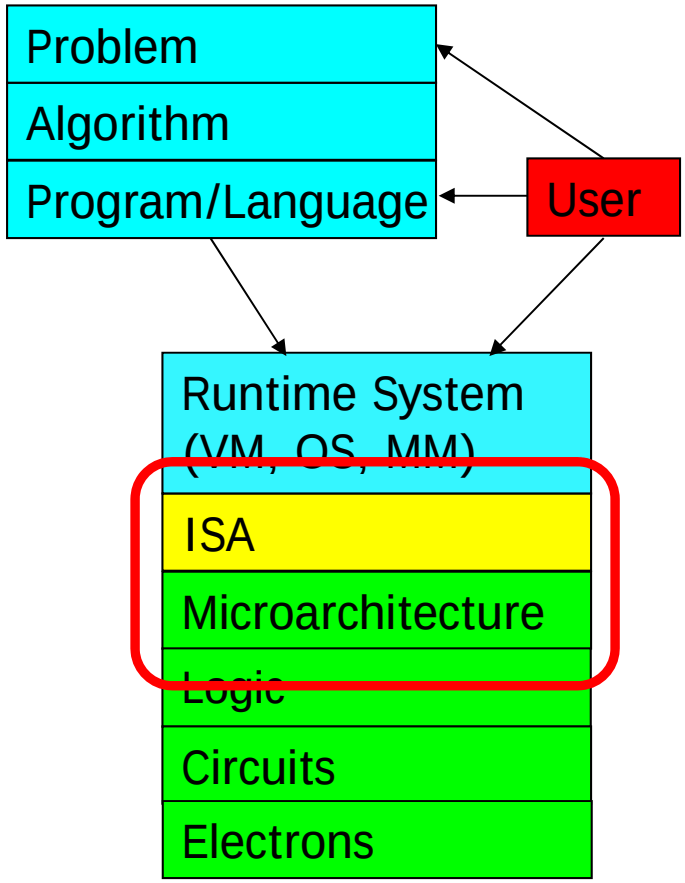
Each of these application areas has a very different set of characteristics. Each application area demands a different set of tradeoffs to be made in specifying the microprocessor to do the job.

Why Is It (Somewhat) Art?



- We do not (fully) know the future (applications, users, market)

Why Is It (Somewhat) Art?



- And, the future is not constant (it changes)!

Analog from Macro-Architecture

- Macro-Architecture: New FCI building

Many Different ISAs Over Decades

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM

- What are the fundamental differences?
  - E.g., how instructions are specified and what they do
  - E.g., how complex are the instructions

Tradeoffs: Soul of Computer Architecture

- ISA-level tradeoffs
- Microarchitecture-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software
- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
  - *Why art?*

Readings

- P&H, Chapter 4, Sections 4.1-4.4
- P&P, revised Appendix A – LC3b datapath and microprogrammed operation

- P&P Chapter 5: LC-3 ISA
- P&P, revised Appendix A – LC3b ISA

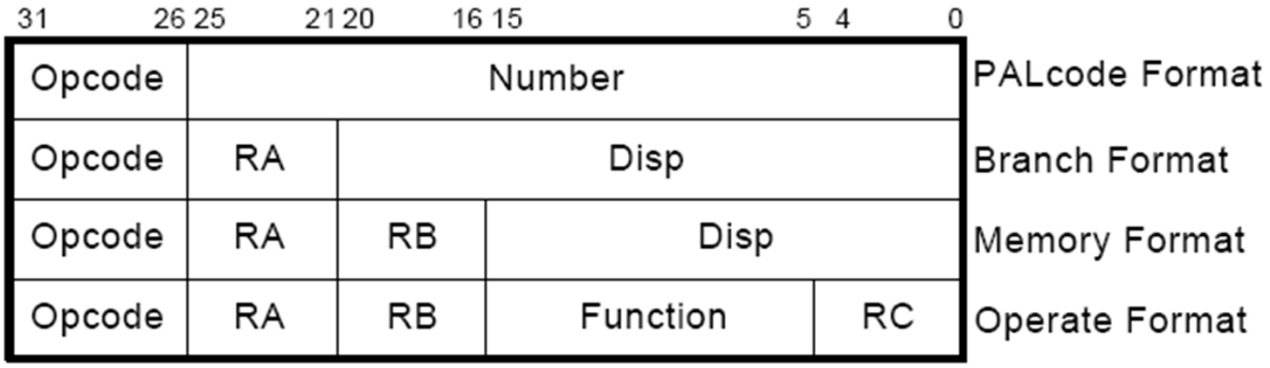
ISA Principles and Tradeoffs

WHAT IS AN INSTRUCTION?

Instruction

- Basic element of the HW/SW interface
- Consists of
  - opcode: what the instruction does
  - operands: who it is to do it to

  - Example from Alpha ISA:



ARM

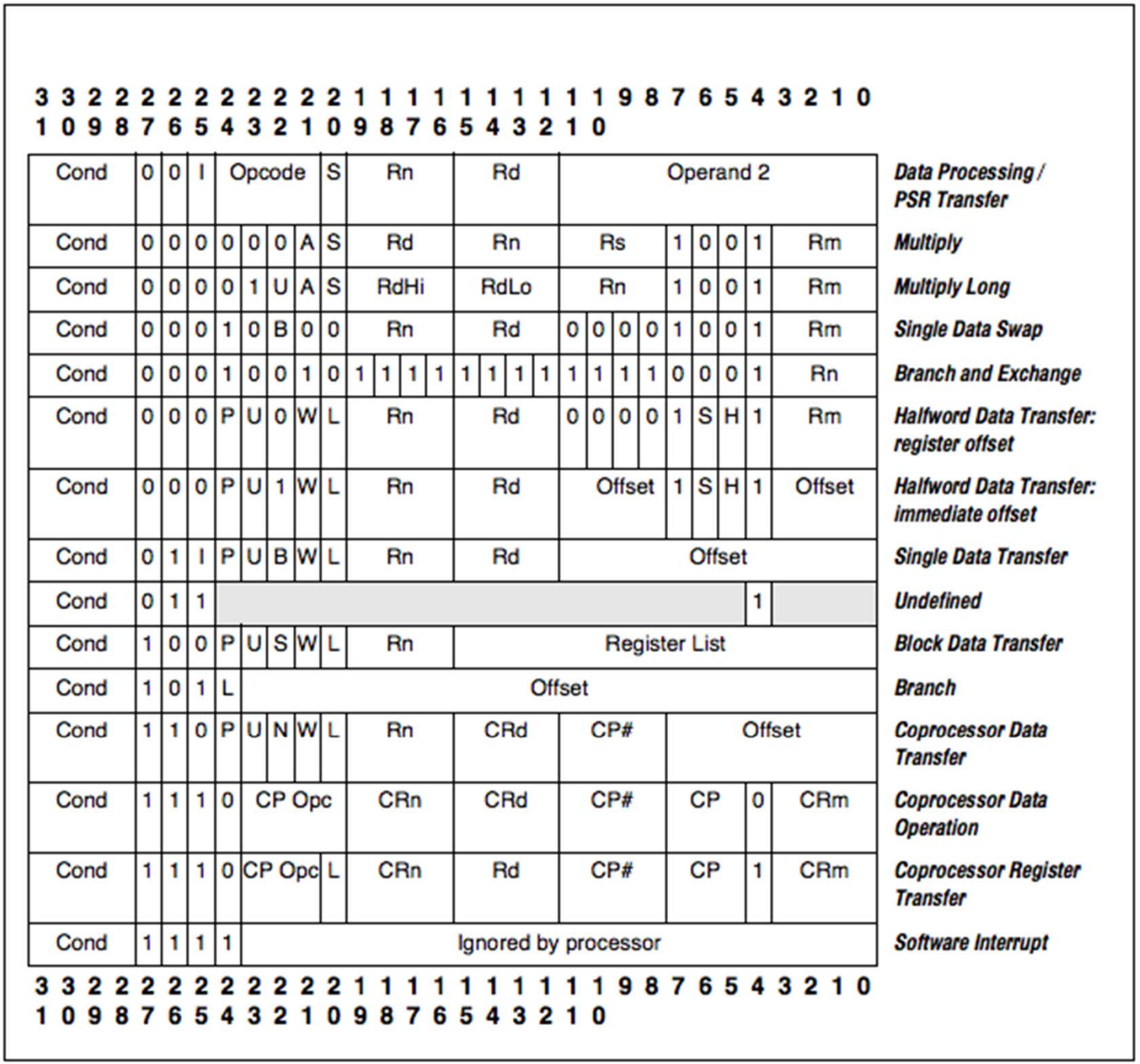
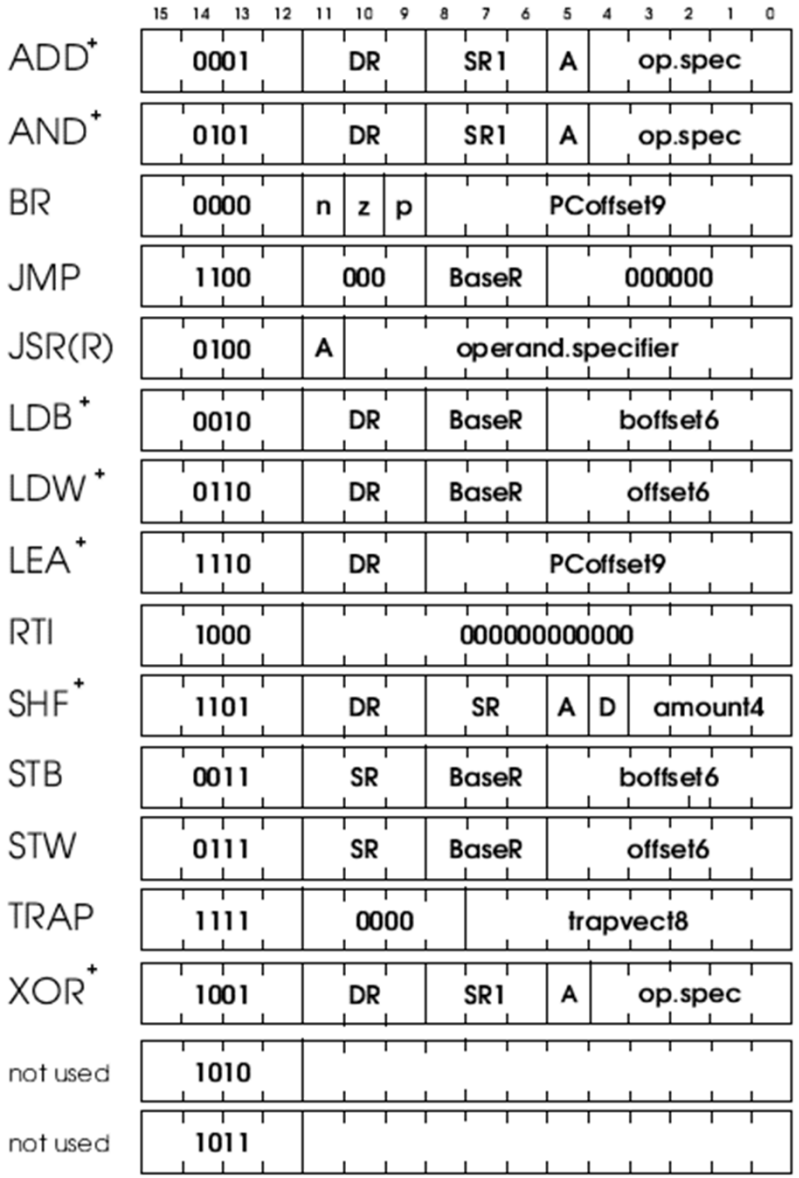


Figure 4-1: ARM instruction set formats

Set of Instructions, Encoding, and Spec

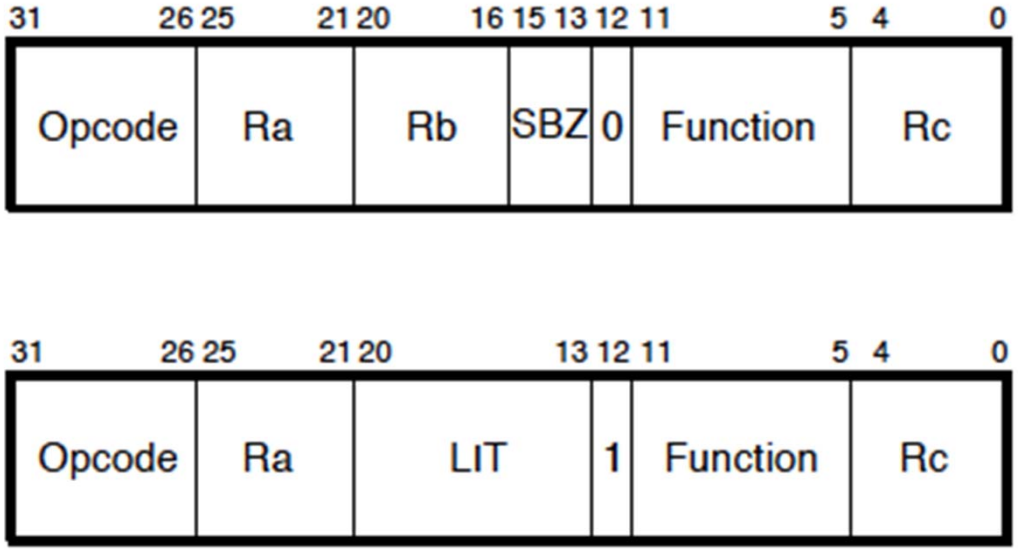


- Example from LC-3b ISA
  - http://www.ece.utexas.edu/~patt/11s.460N/handouts/new_byte.pdf
- x86 Manual

- Why unused instructions?
- Aside: concept of “bit steering”
  - A bit in the instruction determines the interpretation of other bits

Bit Steering in Alpha

Figure 3–4: Operate Instruction Format



If bit <12> of the instruction is 0, the Rb field specifies a source register operand.

If bit <12> of the instruction is 1, an 8-bit zero-extended literal constant is formed by bits <20:13> of the instruction. The literal is interpreted as a positive integer between 0 and 255 and is zero-extended to 64 bits. Symbolically, the integer Rbv operand is formed as follows:

ELEMENTS OF ISA?

1. Instruction processing style

- Specifies the number of “operands” an instruction “operates” on and how it does so
- 0, 1, 2, 3 address machines
  - 0-address: stack machine (push A, pop A, op)
  - 1-address: accumulator machine (ld A, st A, op A)
  - 2-address: 2-operand machine (one is both source and dest)
  - 3-address: 3-operand machine (source and dest are separate)
- Tradeoffs? See your homework question
  - Larger operate instructions vs. more executed operations
  - Code size vs. execution time vs. on-chip memory space

An Example: Stack Machine

+ Small instruction size (no operands needed for operate instructions)

- Simpler logic
- Compact code

+ Efficient procedure calls: all parameters on stack

- No additional cycles for parameter passing

-- Computations that are not easily expressible with “postfix notation” are difficult to map to stack machines

- Cannot perform operations on many values at the same time (only top N values on the stack at the same time)
- Not flexible

An Example: Stack Machine Operation

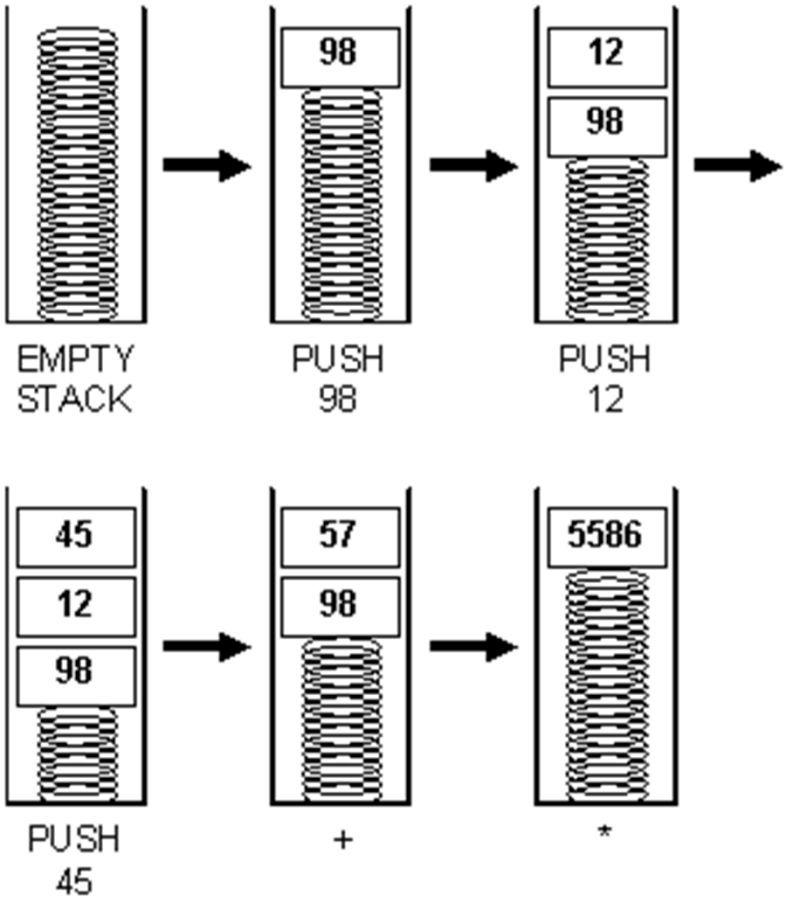


Figure 3.2 -- An example stack machine.

Koopman, "Stack Computers: The New Wave," 1989. http://www.ece.cmu.edu/~koopman/stack_computers/sec3_2.html

Other Examples

- PDP-11: A 2-address machine
  - PDP-11 ADD: 4-bit opcode, 2 6-bit operand specifiers
  - Why? Limited bits to specify an instruction
  - Disadvantage: One source operand is always clobbered with the result of the instruction
    - *How do you ensure you preserve the old value of the source?*

- X86: A 2-address (memory/memory) machine
- Alpha: A 3-address (load/store) machine
- MIPS?
- ARM?

2. Data types

- Definition: Representation of information for which there are instructions that operate on the representation
- Integer, floating point, character, binary, decimal, BCD
- Doubly linked list, queue, string, bit vector, stack
  - VAX: INSQUEUE and REMQUEUE instructions on a doubly linked list or queue; FINDFIRST
  - Digital Equipment Corp., "VAX11 780 Architecture Handbook," 1977.
  - X86: SCAN opcode operates on character strings; PUSH/POP

Data Type Tradeoffs

- What is the benefit of having more or high-level data types in the ISA?
- What is the disadvantage?

- Think compiler/programmer vs. microarchitect

- Concept of semantic gap
  - Data types coupled tightly to the semantic level, or complexity of instructions

- Example: Early RISC architectures vs. Intel 432
  - Early RISC: Only integer data type
  - Intel 432: Object data type, capability based machine

3. Memory organization

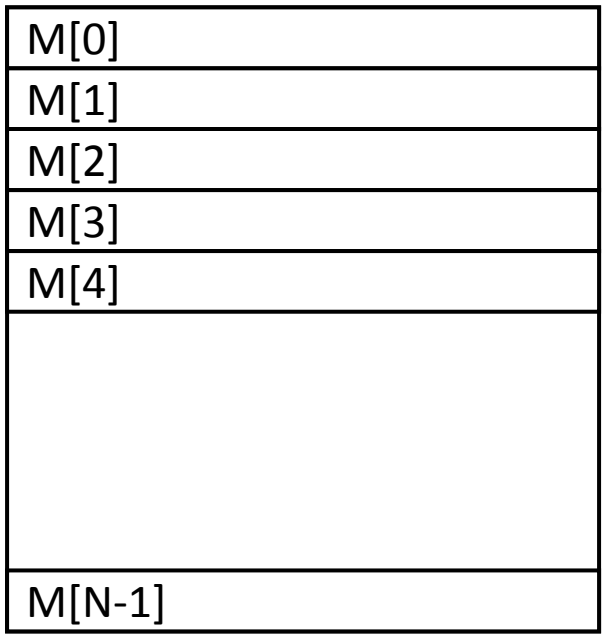
- Address space: How many uniquely identifiable locations in memory
- Addressability: How much data does each uniquely identifiable location store
  - Byte addressable: most ISAs, characters are 8 bits
  - Bit addressable: Burroughs 1700. Why?
  - 64-bit addressable: Some supercomputers. Why?
  - 32-bit addressable: First Alpha
  - Food for thought
    - How do you add 2 32-bit numbers with only byte addressability?
    - How do you add 2 8-bit numbers with only 32-bit addressability?
    - Big endian vs. little endian? MSB at low or high byte.
- Support for virtual memory

4. Registers

- How many
- Size of each register

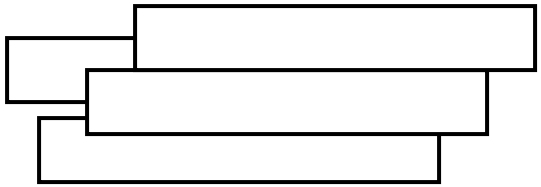
- Why is having registers a good idea?
  - Because programs exhibit a characteristic called data locality
  - A recently produced/accessed value is likely to be used more than once (temporal locality)
    - Storing that value in a register eliminates the need to go to memory each time that value is needed

Programmer Visible (Architectural) State



Memory

array of storage locations
indexed by an address



Registers

- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Program Counter

memory address
of the current instruction

Instructions (and programs) specify how to transform the values of programmer visible state

Aside: Programmer Invisible State

- Microarchitectural state
- Programmer cannot access this directly

- E.g. cache state
- E.g. pipeline registers

Evolution of Register Architecture

- Accumulator
  - a legacy from the “adding” machine days
- Accumulator + address registers
  - need register indirection
  - initially address registers were special-purpose, i.e., can only be loaded with an address for indirection
  - eventually arithmetic on addresses became supported
- General purpose registers (GPR)
  - all registers good for all purposes
  - grew from a few registers to 32 (common for RISC) to 128 in Intel IA-64

Instruction Classes

- Operate instructions
  - Process data: arithmetic and logical operations
  - Fetch operands, compute result, store result
  - Implicit sequential control flow
- Data movement instructions
  - Move data between memory, registers, I/O devices
  - Implicit sequential control flow
- Control flow instructions
  - Change the sequence of instructions that are executed

5. Addressing Mode

- Load/store vs. memory architectures
  - Load/store architecture: operate instructions operate only on registers
    - E.g., MIPS, ARM and many RISC ISAs
  - Memory architecture: operate instructions can operate on memory locations
    - E.g., x86, VAX and many CISC ISAs

What Are the Elements of An ISA?

- Addressing modes specify how to obtain the operands
  - Displacemnt mov ax, [1000]

    use immediate value as address
  - Register Indirect: mov ax, [esi]

    use register as address
  - Displaced or based: mov ax, [bx][si]

    use base register and index register as address
  - Indexed: mov ax, var1[esi]

    use as address

What Are the Benefits of Different Addressing Modes?

- Another example of programmer vs. microarchitect tradeoff

- Advantage of more addressing modes:
  - Enables better mapping of high-level constructs to the machine: some accesses are better expressed with a different mode → reduced number of instructions and code size
    - Think array accesses (autoincrement mode)
    - Think indirection (pointer chasing)
    - Sparse matrix accesses

- Disadvantage:
  - More work for the compiler
  - More work for the microarchitect

5. 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 vs. hardware complexity
  - Performance? Energy? Smaller code vs. ease of decode

6. Uniform Decode

- Uniform decode: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - Same with operand specifiers, immediate values, ...
  - Many "RISC" ISAs: Alpha, MIPS, 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

A Note on Length and Uniformity

- Uniform decode usually goes with fixed length

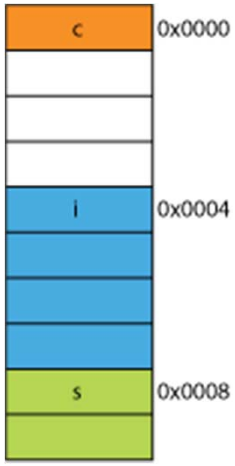
- In a variable length ISA, uniform decode can be a property of instructions of the same length
  - It is hard to think of it as a property of instructions of different lengths

7. Aligned vs. Unaligned access

- The CPU ALWAYS reads at it's word size (4-bytes on a 32-bit processor), so when you do an unaligned address access --on a processor that supports it-- the processor is going to read multiple words. The CPU will read each word of memory that your requested address straddles. This causes an amplification of up to 2x the number of memory transactions required to access the requested data.

- Because of this, it can very easily be slower to read two bytes than four. For example, say you have a struct in memory that looks like this:

```
struct mystruct {
    char c;  // one byte
    int i;   // four bytes
    short s; // two bytes
}
```



- X86 handle Unaligned access, and MIPs deal only with Aligned access otherwise, it will give address exception

8. Complex vs. Simple Instructions

- Complex instruction: An instruction does a lot of work, e.g. many operations
  - Insert in a doubly linked list
  - String copy

- Simple instruction: An instruction does small amount of work, it is a primitive using which complex operations can be built
  - Add
  - XOR
  - Multiply

Complex vs. Simple Instructions

- Advantages of Complex instructions
  - + Denser encoding → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  - + Simpler compiler: no need to optimize small instructions as much

- Disadvantages of Complex Instructions
  - - Larger chunks of work → compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
  - - More complex hardware → translation from a high level to control signals and optimization needs to be done by hardware

ISA-level Tradeoffs: Semantic Gap

- Where to place the ISA? Semantic gap
  - Closer to high-level language (HLL) → Small semantic gap, complex instructions
  - Closer to hardware control signals? → Large semantic gap, simple instructions

- RISC vs. CISC machines
  - RISC: Reduced instruction set computer
  - CISC: Complex instruction set computer
    - FFT, QUICKSORT, POLY, FP instructions?
    - VAX INDEX instruction (array access with bounds checking)

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
      - 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*

How High or Low Can You Go?

- Very large semantic gap
  - Each instruction specifies the complete set of control signals in the machine
  - Compiler generates control signals
  - Open microcode (John Cocke, circa 1970s)
    - Gave way to optimizing compilers

- Very small semantic gap
  - ISA is (almost) the same as high-level language
  - Java machines, LISP machines, object-oriented machines, capability-based machines

A Note on RISC vs. CISC

- Usually, ...

- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes

- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes

ISA EVOLUTION

A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day

- Examples:
  - Limited on-chip and off-chip memory size
  - Limited compiler optimization technology
  - Limited memory bandwidth
  - Need for specialization in important applications (e.g., MMX)

- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
  - Concept of dynamic/static interface
  - Contrast it with hardware/software interface

Effect of Translation

- One can translate from one ISA to another *ISA* to change the semantic gap tradeoffs

- Examples
  - Intel's and AMD's x86 implementations translate x86 instructions into programmer-invisible microoperations (simple instructions) in hardware
  - Transmeta's x86 implementations translated x86 instructions into "secret" VLIW instructions in software (code morphing software)

- Think about the tradeoffs