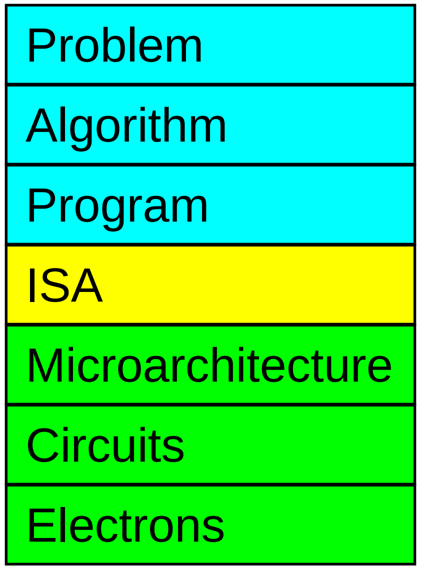
18-447
Computer Architecture
Lecture 3: ISA Tradeoffs

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 1/17/2014

Design Point

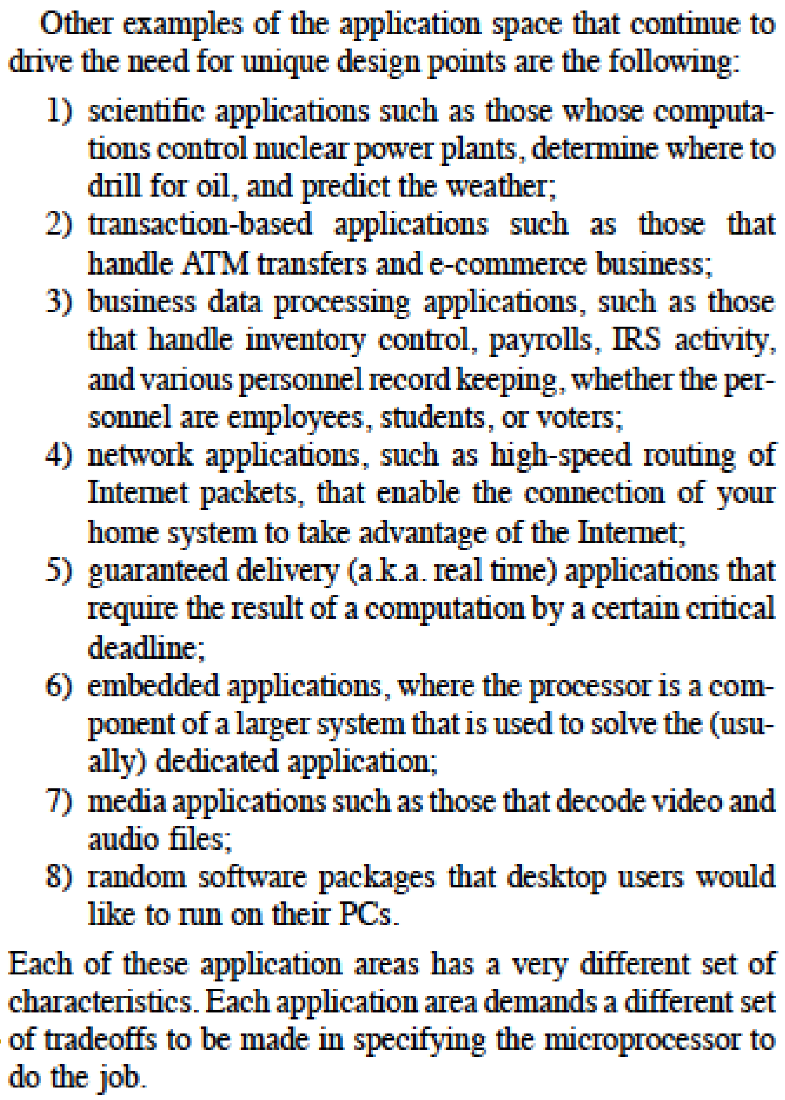
- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Considerations
  - Cost
  - Performance
  - Maximum power consumption
  - Energy consumption (battery life)
  - Availability
  - Reliability and Correctness
  - Time to Market



- 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:

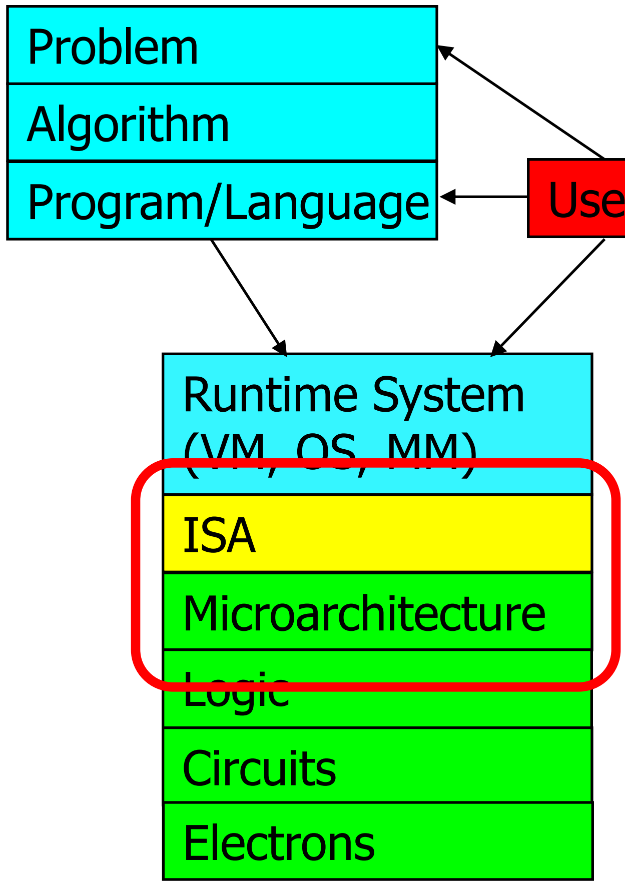
1) scientific applications such as those whose computations control nuclear power plants, determine where to drill for oil, and predict the weather;
2) transaction-based applications such as those that handle ATM transfers and e-commerce business;
3) business data processing applications, such as those 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.

Tradeoffs: Soul of Computer Architecture

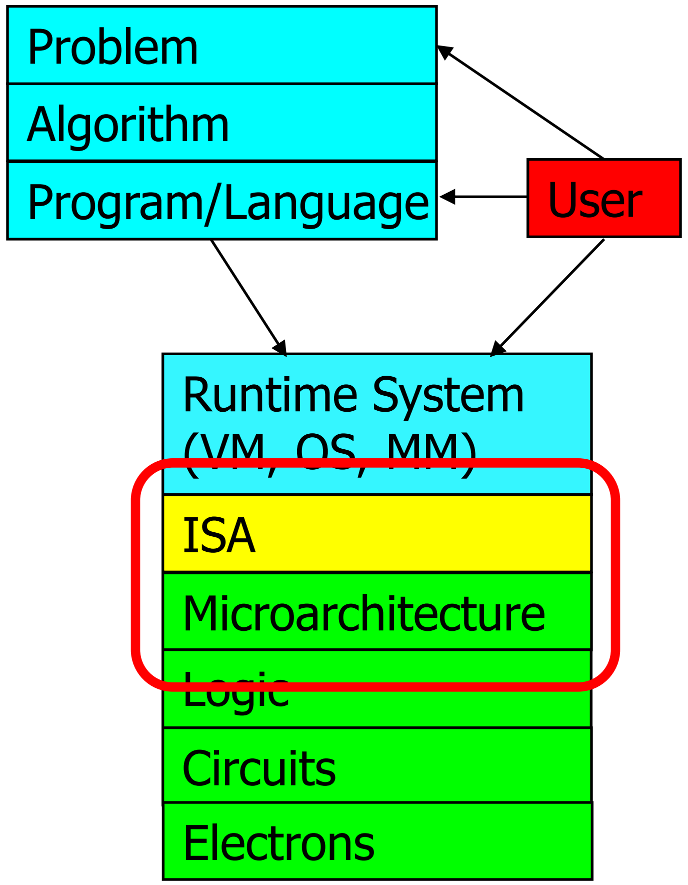
- ISA-level tradeoffs
- Microarchitecture-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software
- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
  - *Why art?*

Why Is It (Somewhat) Art?



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

Why Is It (Somewhat) Art?

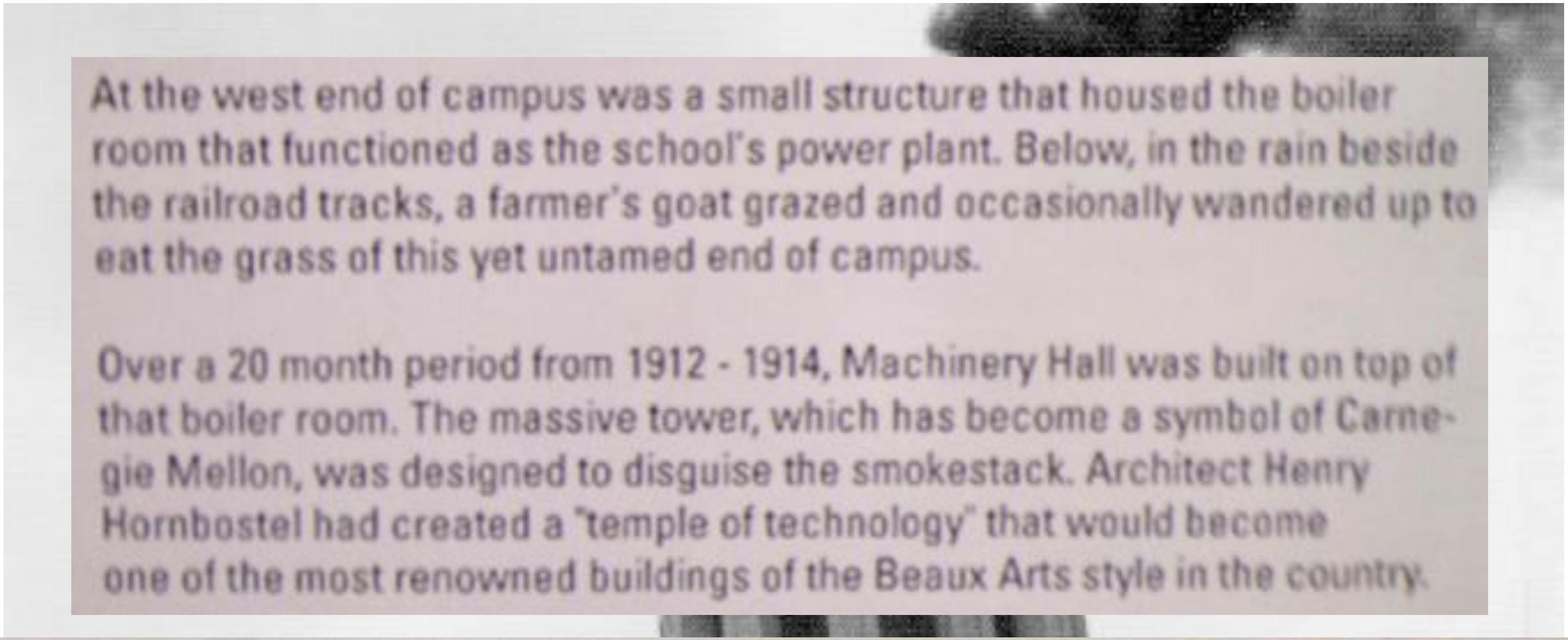


- And, the future is not constant (it changes)!

Analog from Macro-Architecture

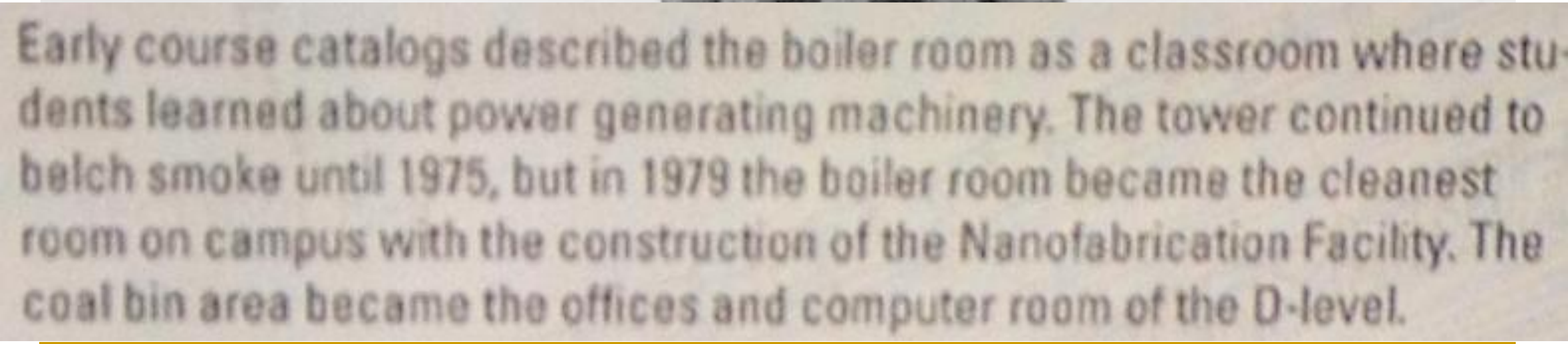
- Future is not constant in macro-architecture, either
- Example: Can a power plant boiler room be later used as a classroom?

Macro-Architecture: Boiler Room



At the west end of campus was a small structure that housed the boiler room that functioned as the school's power plant. Below, in the rain beside the railroad tracks, a farmer's goat grazed and occasionally wandered up to eat the grass of this yet untamed end of campus.

Over a 20 month period from 1912 - 1914, Machinery Hall was built on top of that boiler room. The massive tower, which has become a symbol of Carnegie Mellon, was designed to disguise the smokestack. Architect Henry Hornbostel had created a "temple of technology" that would become one of the most renowned buildings of the Beaux Arts style in the country.



Early course catalogs described the boiler room as a classroom where students learned about power generating machinery. The tower continued to belch smoke until 1975, but in 1979 the boiler room became the cleanest room on campus with the construction of the Nanofabrication Facility. The coal bin area became the offices and computer room of the D-level.

Readings for Next Time

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

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

ISA Principles and Tradeoffs

Many Different ISAs Over Decades

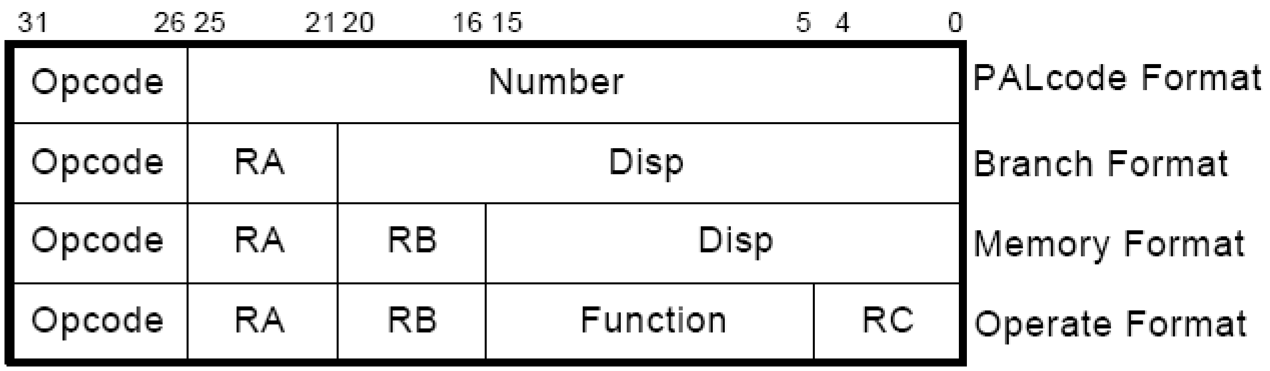
- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM

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

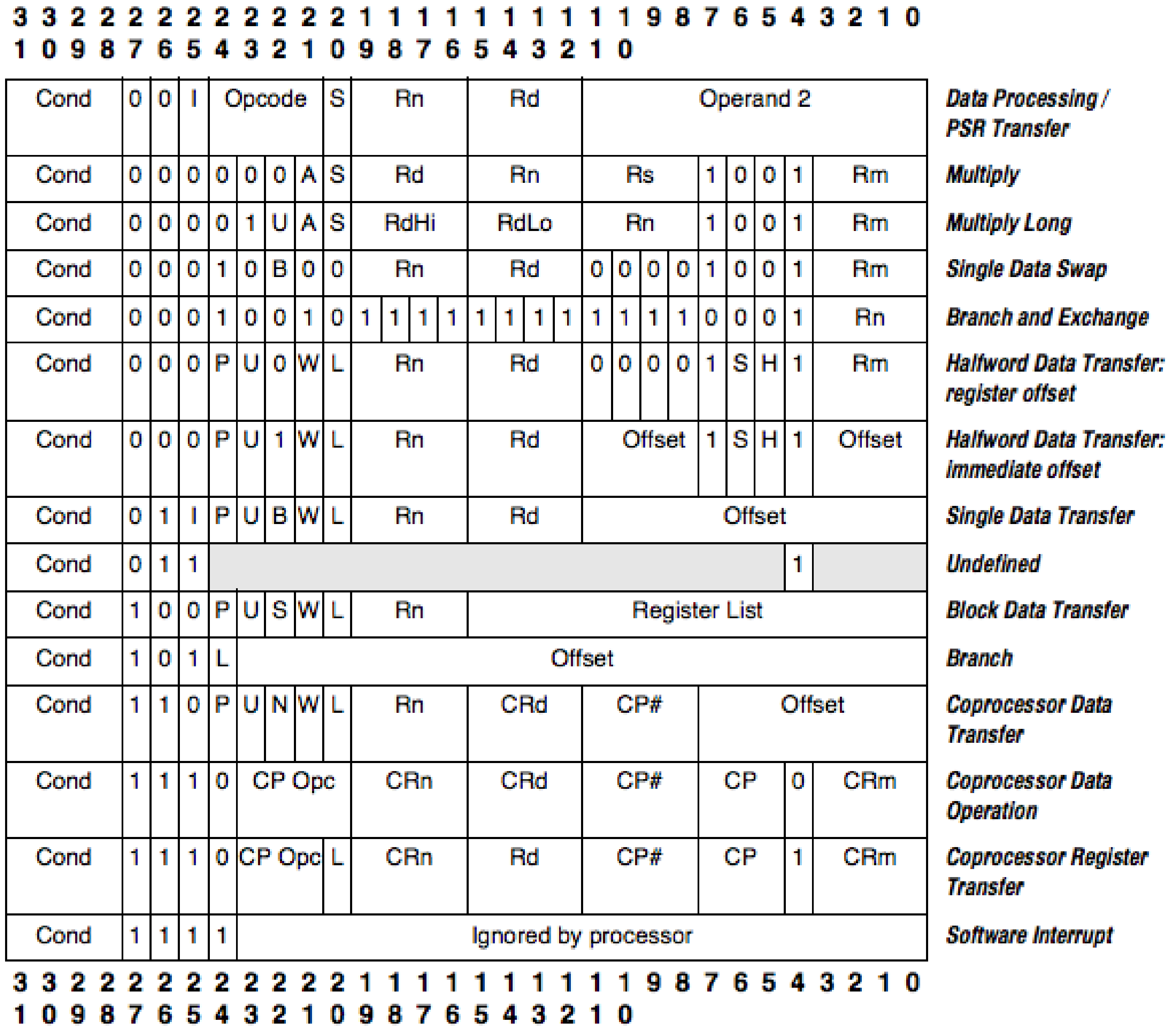
Instruction

- 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:



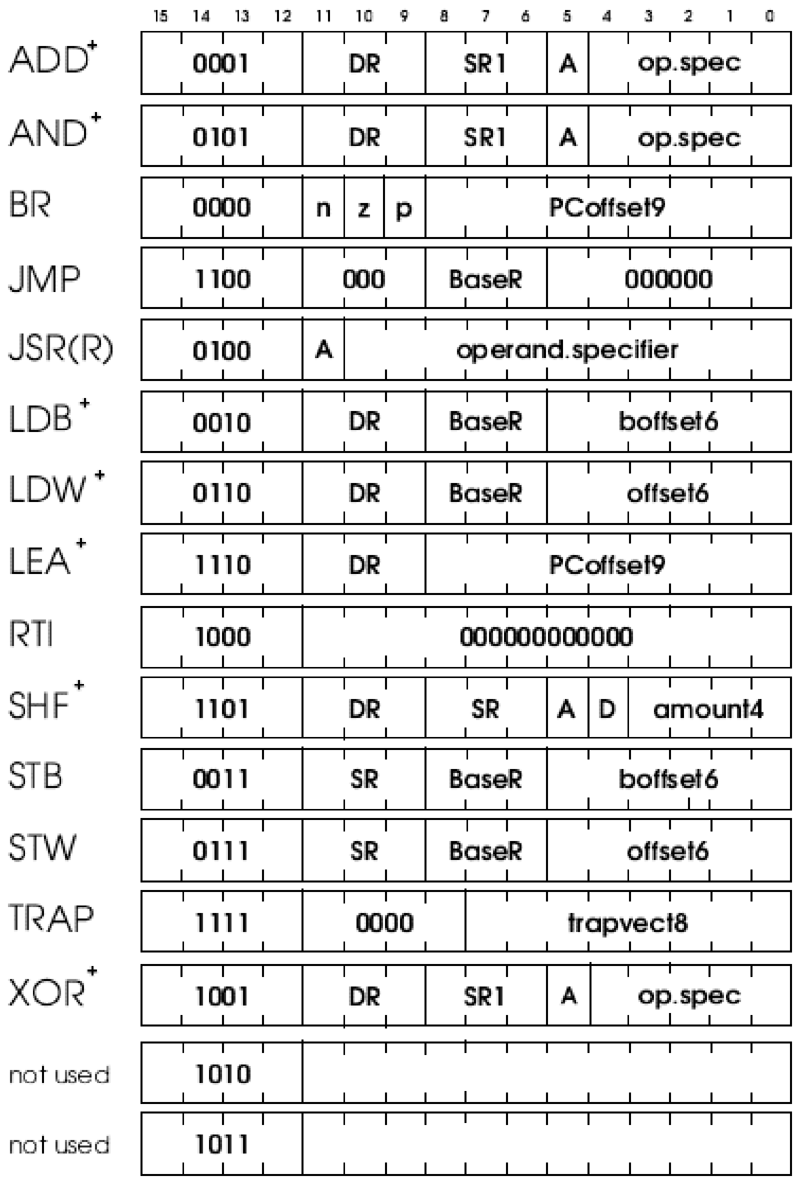
ARM



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cond				0	0	I	Opcode				S	Rn				Rd				Operand 2												*Data Processing / PSR Transfer*
Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				*Multiply*
Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				*Multiply Long*
Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				*Single Data Swap*
Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				*Branch and Exchange*
Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				*Halfword Data Transfer: register offset*
Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				*Halfword Data Transfer: immediate offset*
Cond				0	1	I	P	U	B	W	L	Rn				Rd				Offset												*Single Data Transfer*
Cond				0	1	1																					1					*Undefined*
Cond				1	0	0	P	U	S	W	L	Rn				Register List																*Block Data Transfer*
Cond				1	0	1	L	Offset																								*Branch*
Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset								*Coprocessor Data Transfer*
Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm				*Coprocessor Data Operation*
Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm				*Coprocessor Register Transfer*
Cond				1	1	1	1	Ignored by processor																								*Software Interrupt*
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Figure 4-1: ARM instruction set formats

Set of Instructions, Encoding, and Spec



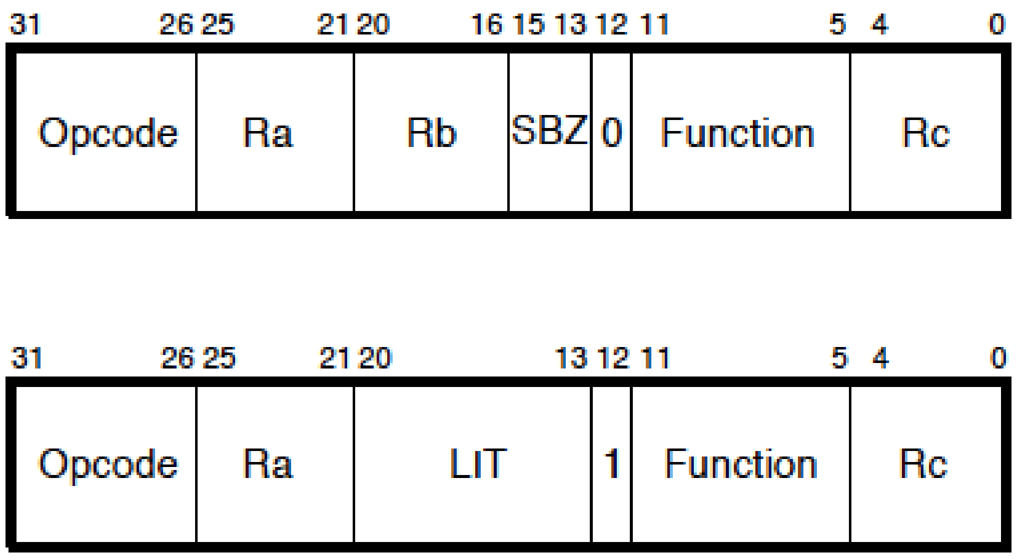
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD*	0001				DR			SR1			A	op.spec				
AND*	0101				DR			SR1			A	op.spec				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR(R)	0100				A	operand.specifier										
LDB*	0010				DR			BaseR			boffset6					
LDW*	0110				DR			BaseR			offset6					
LEA*	1110				DR			PCoffset9								
RTI	1000				000000000000											
SHF*	1101				DR			SR			A	D	amount4			
STB	0011				SR			BaseR			boffset6					
STW	0111				SR			BaseR			offset6					
TRAP	1111				0000				trapvect8							
XOR*	1001				DR			SR1			A	op.spec				
not used	1010															
not used	1011															

- 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:

What Are the Elements of An ISA?

- Instruction sequencing model
  - Control flow vs. data flow
  - Tradeoffs?

- 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 (II)

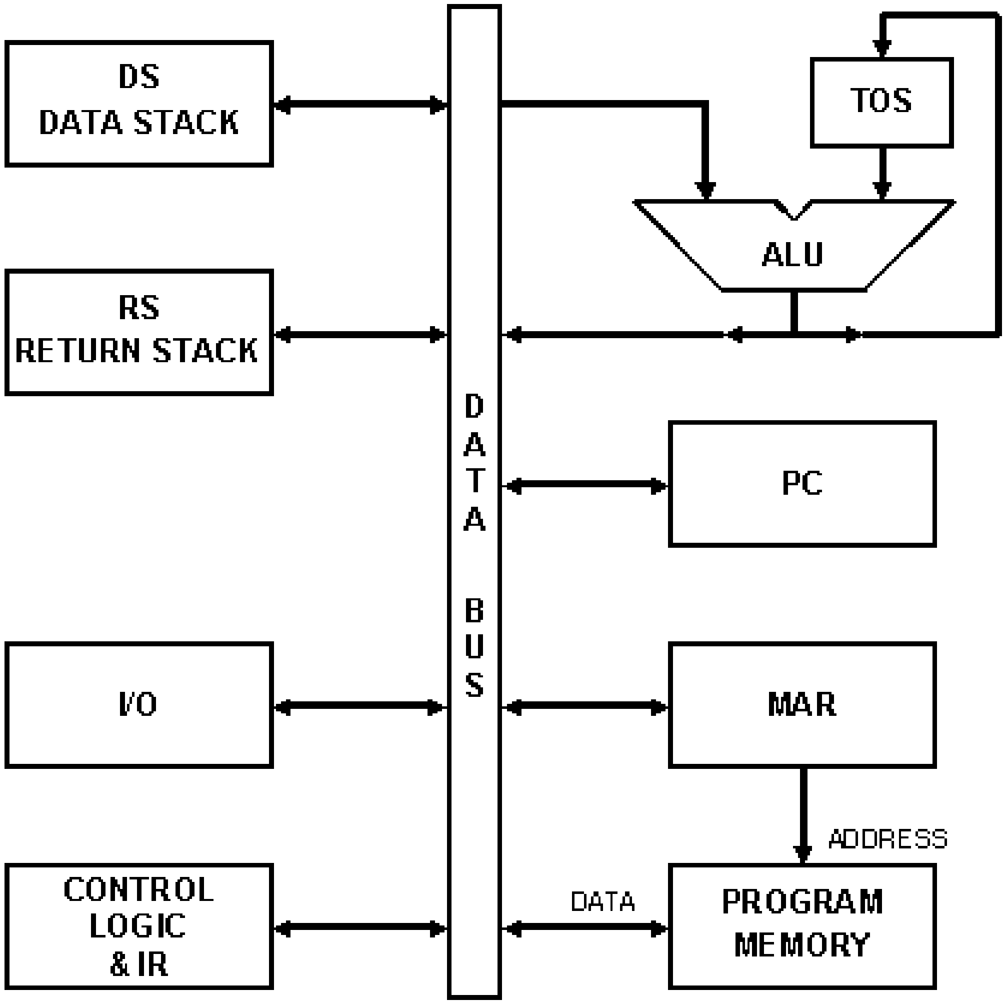


Figure 3.1 -- The canonical stack machine.

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

An Example: Stack Machine Operation

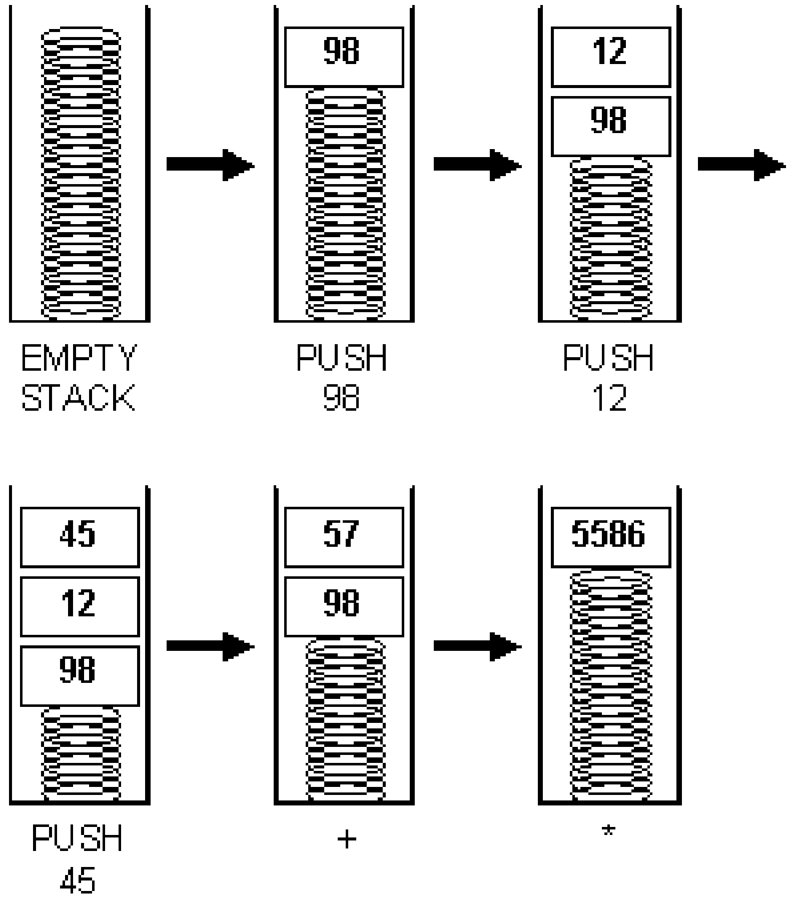


Figure 3.2 -- An example stack machine.

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?

What Are the Elements of An ISA?

- Instructions
  - Opcode
  - Operand specifiers (addressing modes)
    - How to obtain the operand? Why are there different addressing modes?

- 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

What Are the Elements of An ISA?

- 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

Some Historical Readings

- If you want to dig deeper

- Wilner, "Design of the Burroughs 1700," AFIPS 1972.

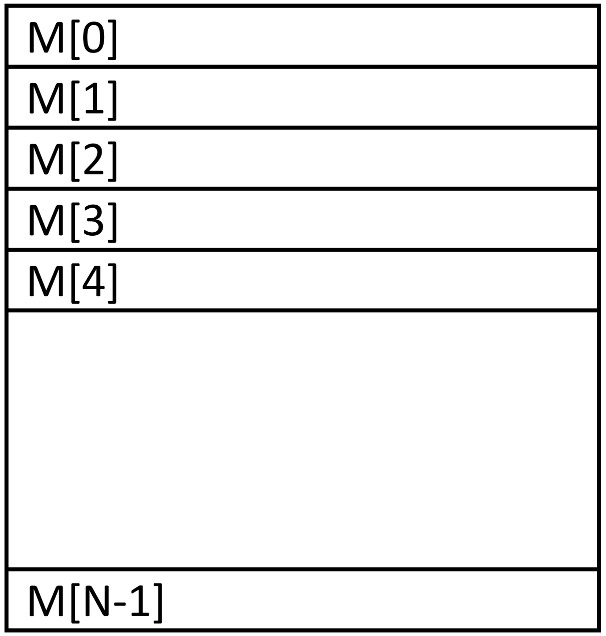
- Levy, "The Intel iAPX 432," 1981.
  - http://www.cs.washington.edu/homes/levy/capabook/Chapter9.pdf

What Are the Elements of An ISA?

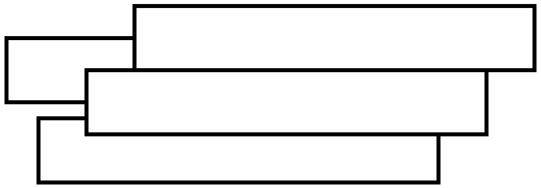
- Registers
  - How many
  - Size of each register

- 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

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

What Are the Elements of An ISA?

- Load/store vs. memory/memory architectures
  - Load/store architecture: operate instructions operate only on registers
    - E.g., MIPS, ARM and many RISC ISAs
  - Memory/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
  - Absolute LW rt, 10000

    use immediate value as address
  - Register Indirect: LW rt, (r_{base})

    use GPR[r_{base}] as address
  - Displaced or based: LW rt, offset(r_{base})

    use offset+GPR[r_{base}] as address
  - Indexed: LW rt, (r_{base}, r_{index})

    use GPR[r_{base}]+GPR[r_{index}] as address
  - Memory Indirect LW rt ((r_{base}))

    use value at M[GPR[r_{base}]] as address
  - Auto inc/decrement LW Rt, (r_{base})

    use GRP[r_{base}] as address, but inc. or dec. GPR[r_{base}] each time

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

ISA Orthogonality

- Orthogonal ISA:
  - All addressing modes can be used with all instruction types
  - Example: VAX
    - (~13 addressing modes) x (>300 opcodes) x (integer and FP formats)

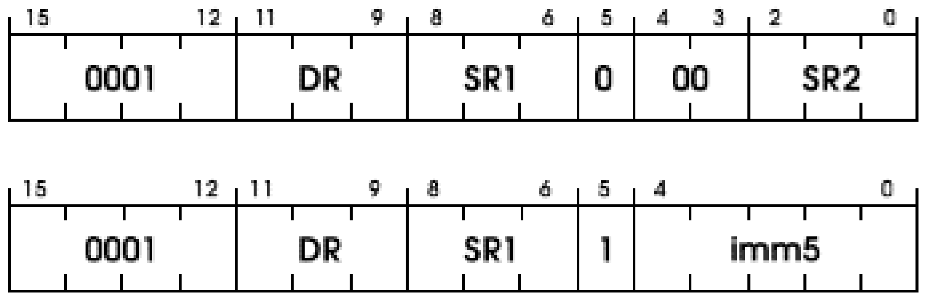
- Who is this good for?
- Who is this bad for?

Is the LC-3b ISA Orthogonal?

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD*	0001				DR			SR1			A	op.spec				
AND*	0101				DR			SR1			A	op.spec				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR(R)	0100				A	operand.specifier										
LDB*	0010				DR			BaseR			boffset6					
LDW*	0110				DR			BaseR			offset6					
LEA*	1110				DR			PCoffset9								
RTI	1000				000000000000											
SHF*	1101				DR			SR			A	D	amount4			
STB	0011				SR			BaseR			boffset6					
STW	0111				SR			BaseR			offset6					
TRAP	1111				0000				trapvect8							
XOR*	1001				DR			SR1			A	op.spec				
not used	1010															
not used	1011															

LC-3b: Addressing Modes of ADD

Encodings



Operation

```
if (bit[5] == 0)
   DR = SR1 + SR2;
else
   DR = SR1 + SEXT(imm5);
setcc();
```

LC-3b: Addressing Modes of of JSR(R)

Encodings

	15–12	11	10–0
JSR	0100	1	PCoffset11

	15–12	11	10–9	8–6	5–0
JSRR	0100	0	00	BaseR	000000

Operation

```
R7 = PC†;
if (bit[11] == 0)
  PC = BaseR;
else
  PC = PC† + LSHF(SEXT(PCoffset11), 1);
```

Description

First, the incremented PC is saved in R7. This is the linkage back to the calling routine. Then, the PC is loaded with the address of the first instruction of the subroutine, causing an unconditional jump to that address. The address of the subroutine is obtained from the base register (if bit[11] is 0), or the address is computed by sign-extending bits [10:0] to 16 bits, left-shifting the result one bit, and then adding this value to the incremented PC (if bit[11] is 1).

What Are the Elements of An ISA?

- How to interface with I/O devices
  - Memory mapped I/O
    - A region of memory is mapped to I/O devices
    - I/O operations are loads and stores to those locations

  - Special I/O instructions
    - IN and OUT instructions in x86 deal with ports of the chip

  - Tradeoffs?
    - Which one is more general purpose?

What Are the Elements of An ISA?

- Privilege modes
  - User vs supervisor
  - Who can execute what instructions?
- Exception and interrupt handling
  - What procedure is followed when something goes wrong with an instruction?
  - What procedure is followed when an external device requests the processor?
  - Vectored vs. non-vectored interrupts (early MIPS)
- Virtual memory
  - Each program has the illusion of the entire memory space, which is greater than physical memory
- Access protection

- We will talk about these later

Another Question

- Does the LC-3b ISA contain complex instructions?

Complex vs. Simple Instructions

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

ISA-level Tradeoffs: Semantic Gap

- Some tradeoffs (for you to think about)

- 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

X86: Small Semantic Gap: String Operations

- An instruction operates on a string
  - Move one string of arbitrary length to another location
  - Compare two strings

- Enabled by the ability to specify repeated execution of an instruction (in the ISA)
  - Using a "prefix" called REP prefix

- Example: REP MOVS instruction
  - Only two bytes: REP prefix byte and MOVS opcode byte (F2 A4)
  - Implicit source and destination registers pointing to the two strings (ESI, EDI)
  - Implicit count register (ECX) specifies how long the string is

X86: Small Semantic Gap: String Operations

REP MOVS **(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|E)SI ← (R|E)SI + 1;
            (R|E)DI ← (R|E)DI + 1;
        ELSE
            (R|E)SI ← (R|E)SI – 1;
            (R|E)DI ← (R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 2;
            (R|E)DI ← (R|E)DI + 2;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 2;
            (R|E)DI ← (R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 4;
            (R|E)DI ← (R|E)DI + 4;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 4;
            (R|E)DI ← (R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 8;
            (R|E)DI ← (R|E)DI + 8;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 8;
            (R|E)DI ← (R|E)DI – 8;
        FI;
FI;
```

How many instructions does this take in ARM and MIPS?

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
      - 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 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