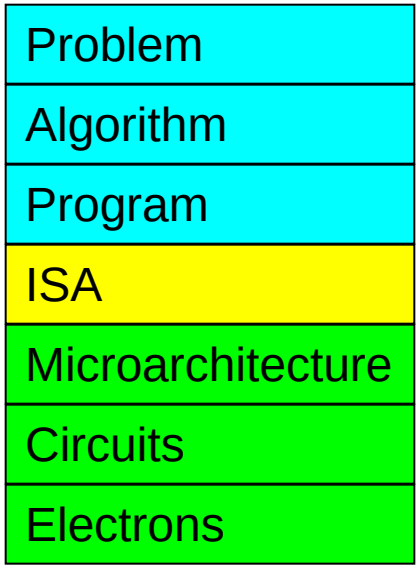
18-447
Computer Architecture
Lecture 3: ISA Tradeoffs

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 1/17/2013

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:

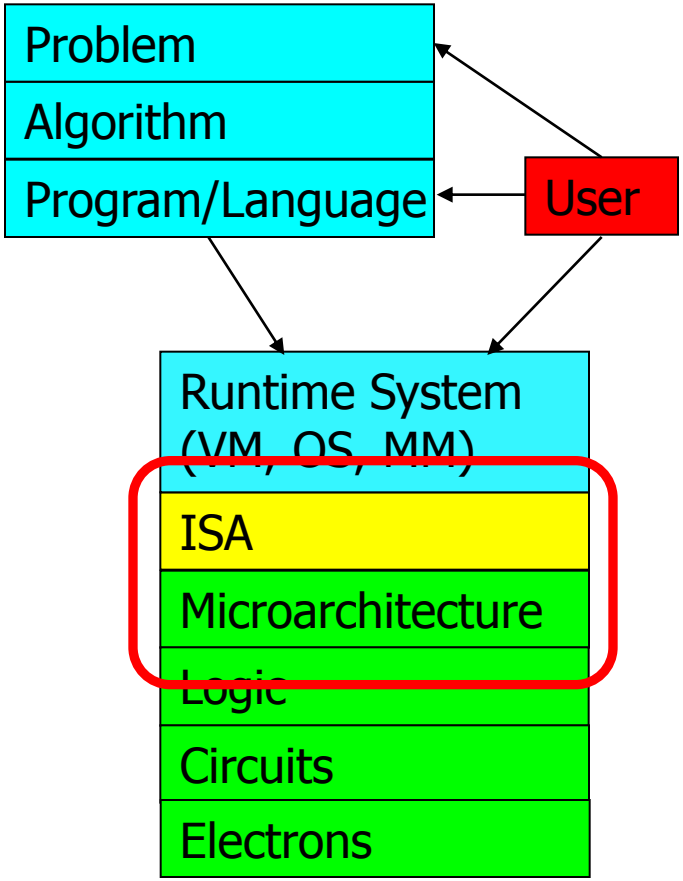
1) scientific applications such as those whose computations control nuclear power plants, determine where to drill for oil, and predict the weather;
2) transaction-based applications such as those that handle ATM transfers and e-commerce business;
3) business data processing applications, such as those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

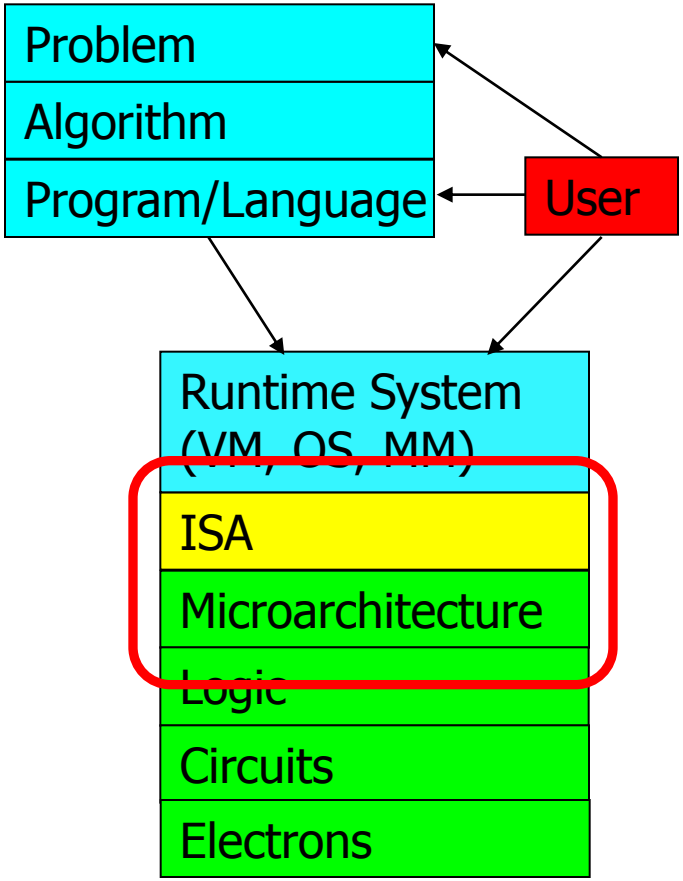
- ISA-level tradeoffs
- Microarchitecture-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software
- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
  - *Why art?*

Why Is It (Somewhat) Art?



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

Why Is It (Somewhat) Art?

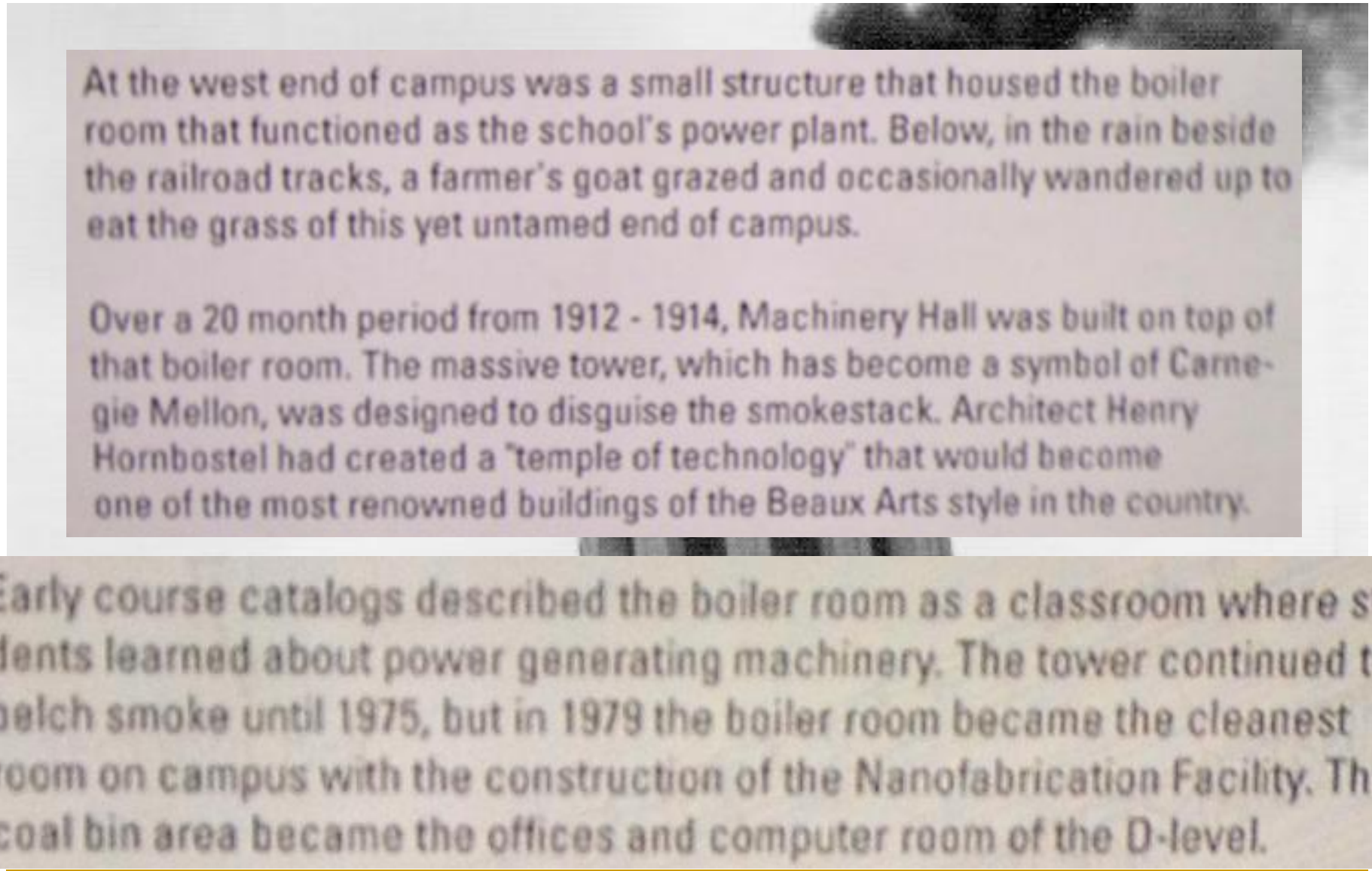


- And, the future is not constant (it changes)!

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

- Optional:
  - P&P Chapter 5: LC-3 ISA

ISA Principles and Tradeoffs

Many Different ISAs Over Decades

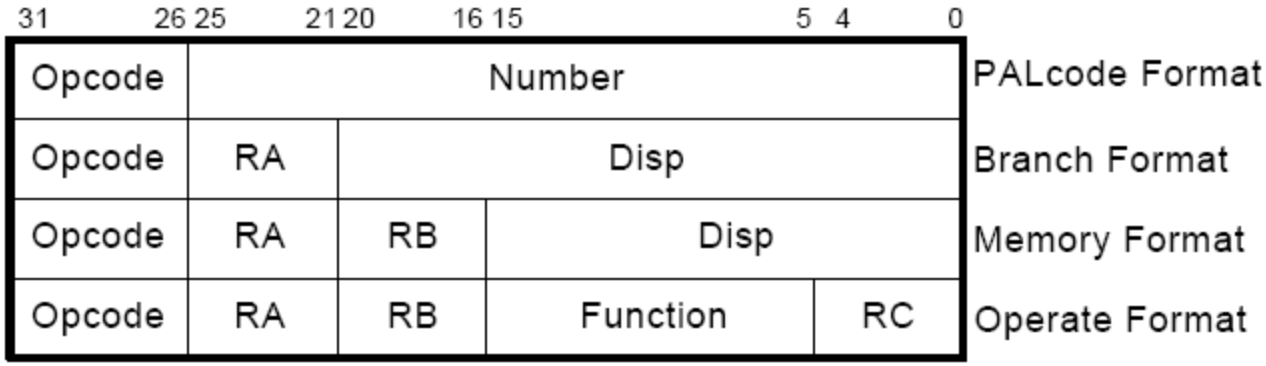
- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM

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

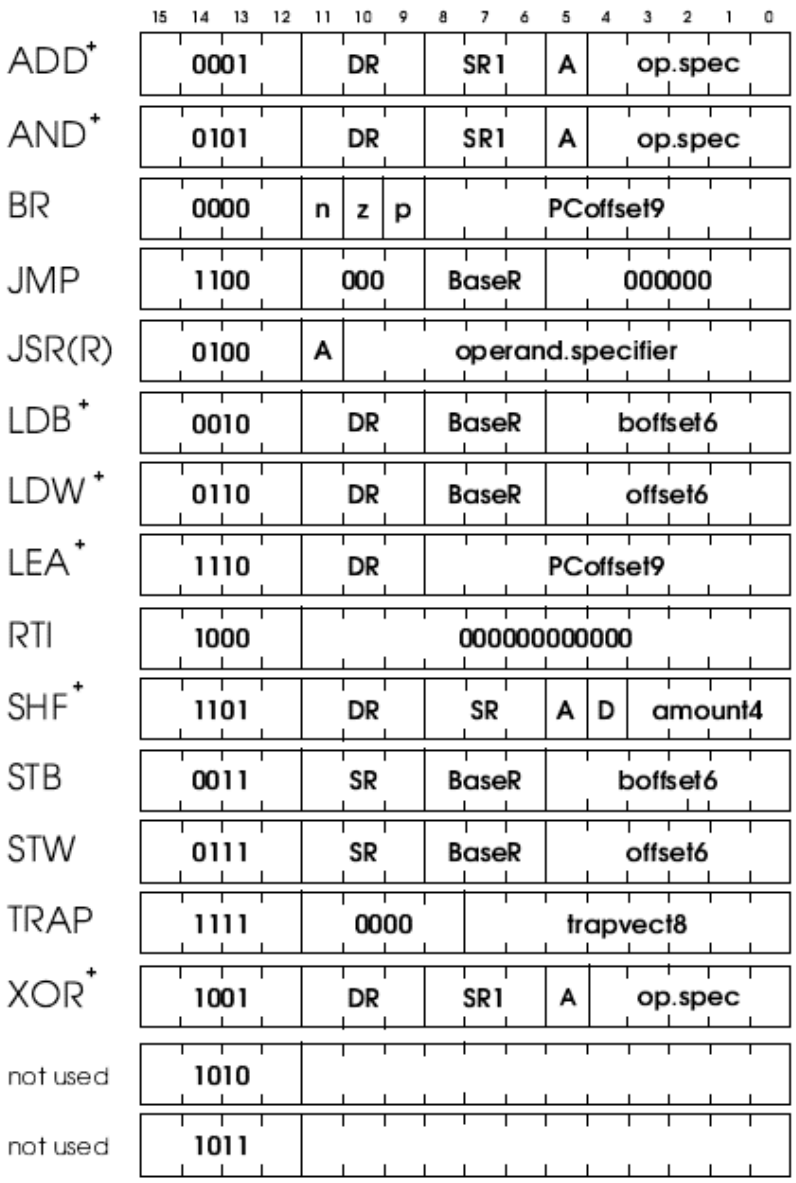
Instruction

- 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:



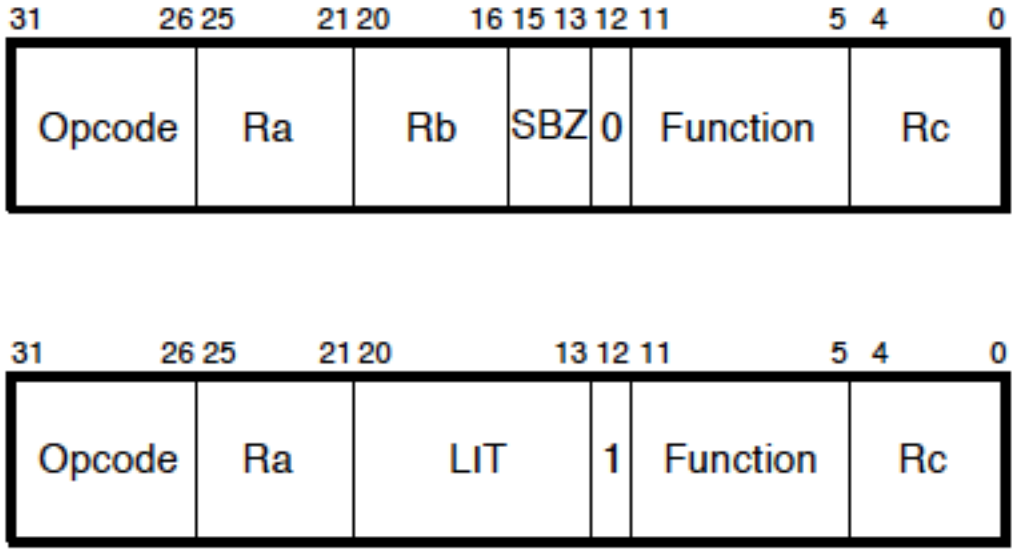
Set of Instructions, Encoding, and Spec



- Example from LC-3b ISA
  - http://www.ece.utexas.edu/~patt/11s.460N/handouts/new_byte.pdf
- x86 Manual
- Aside: concept of "bit steering"
  - A bit in the instruction determines the interpretation of other bits
- Why unused instructions?

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)

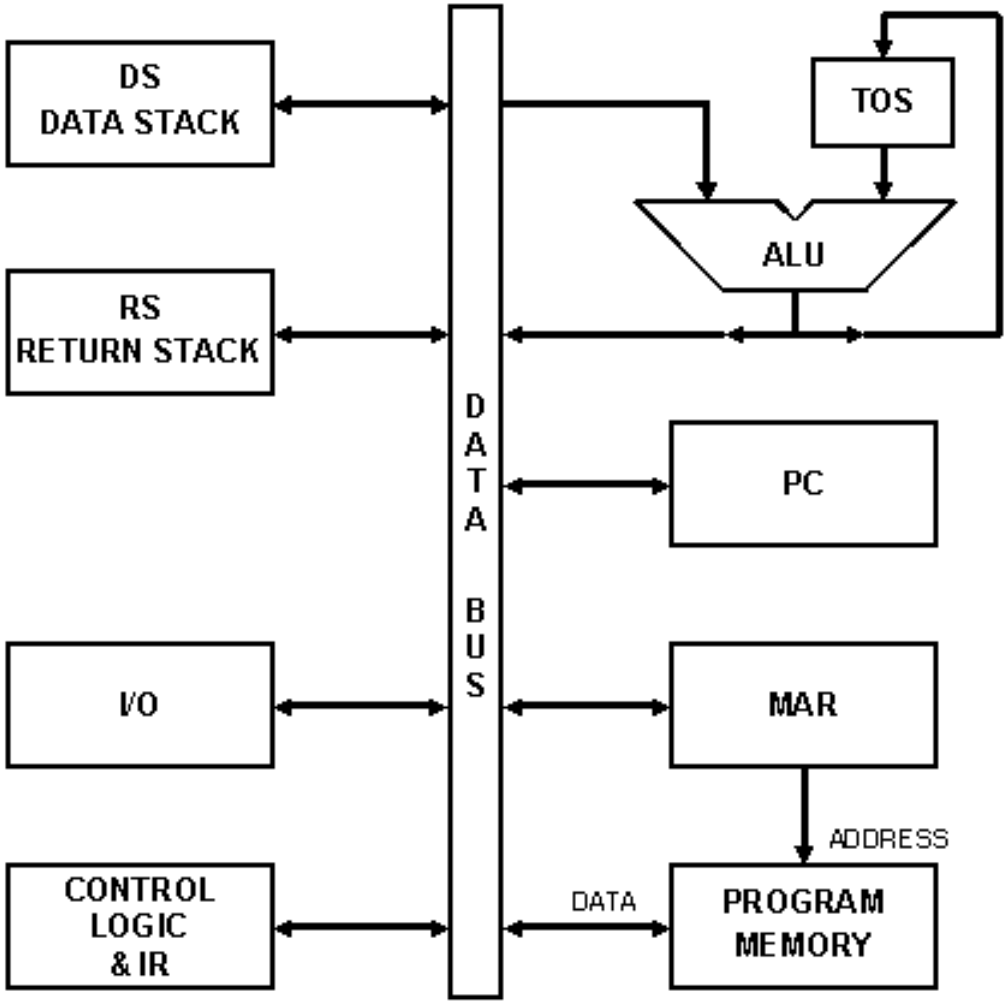


Figure 3.1 -- The canonical stack machine.

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

An Example: Stack Machine Operation

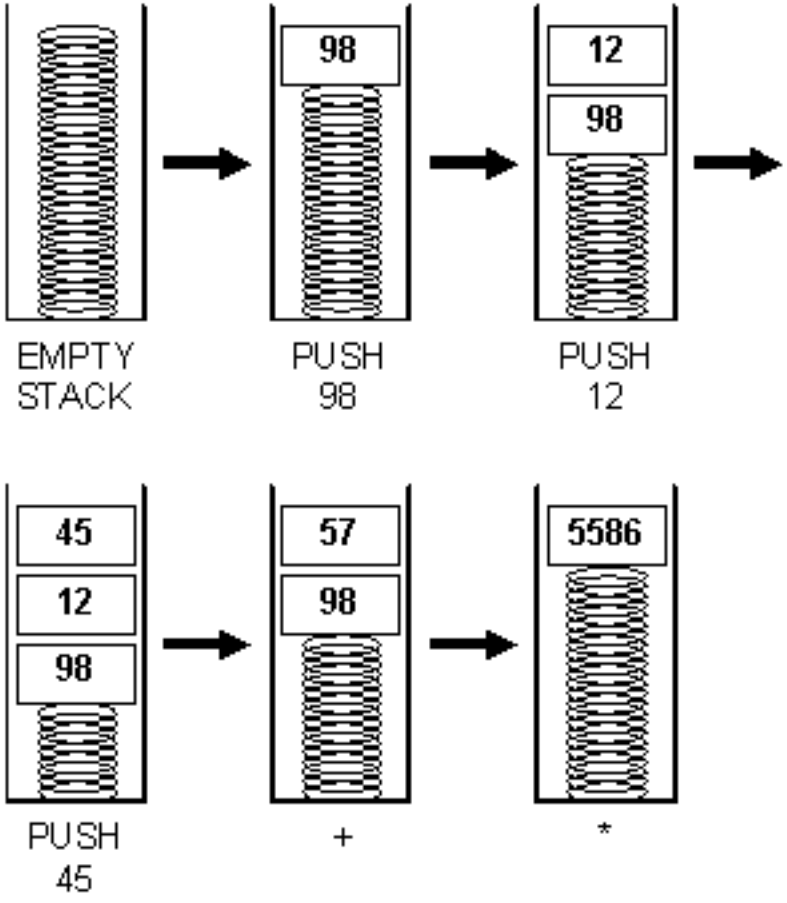


Figure 3.2 -- An example stack machine.

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?

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

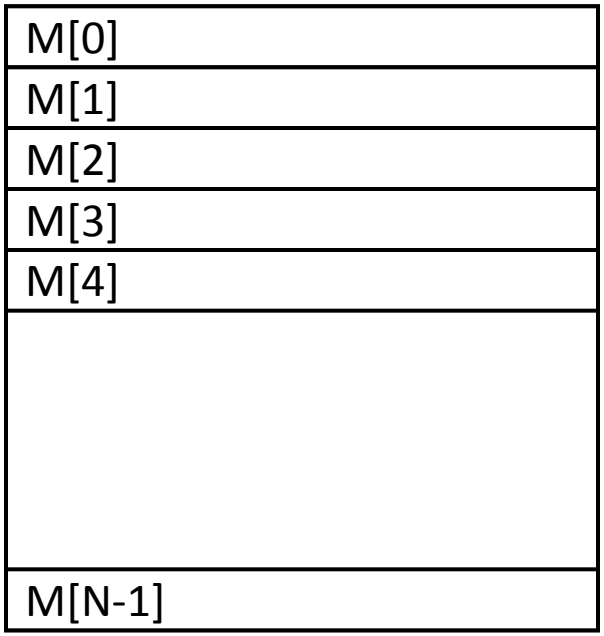
- If you want to dig deeper
- Wilner, "Design of the Burroughs 1700," AFIPS 1972.
- Levy, "The Intel iAPX 432," 1981.
  - http://www.cs.washington.edu/homes/levy/capabook/Chapter9.pdf

What Are the Elements of An ISA?

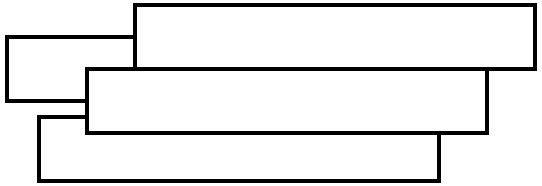
- Registers
  - How many
  - Size of each register

- 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

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

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

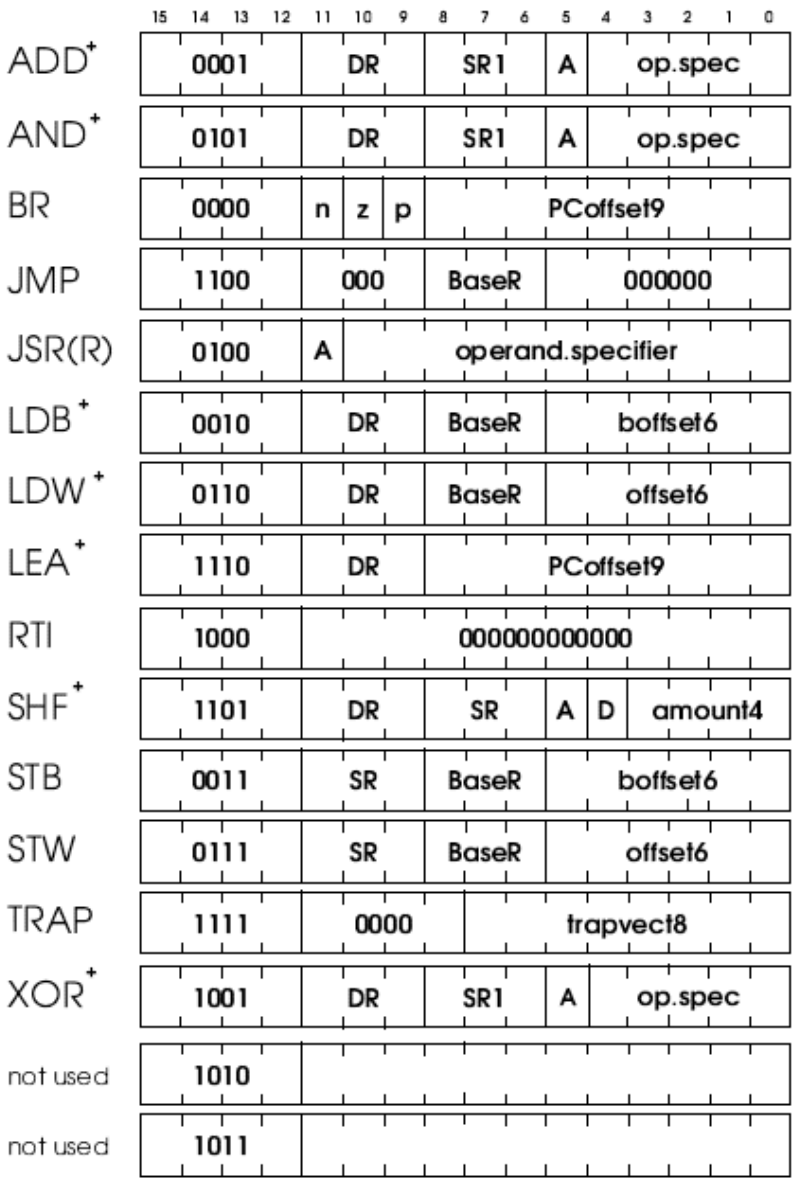
- Disadvantage:
  - More work for the compiler
  - More work for the microarchitect

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)

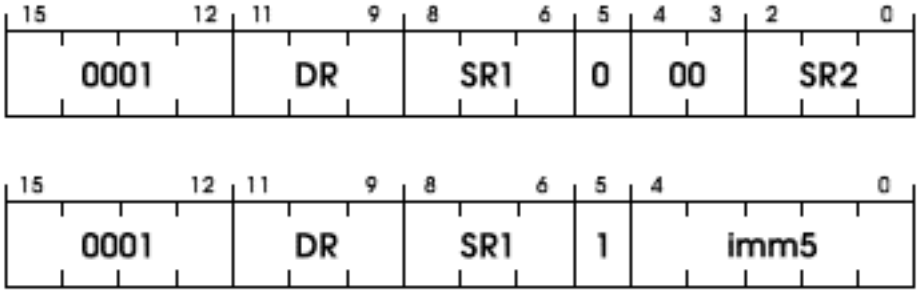
- Who is this good for?
- Who is this bad for?

Is the LC-3b ISA Orthogonal?



LC-3b: Addressing Modes of ADD

Encodings

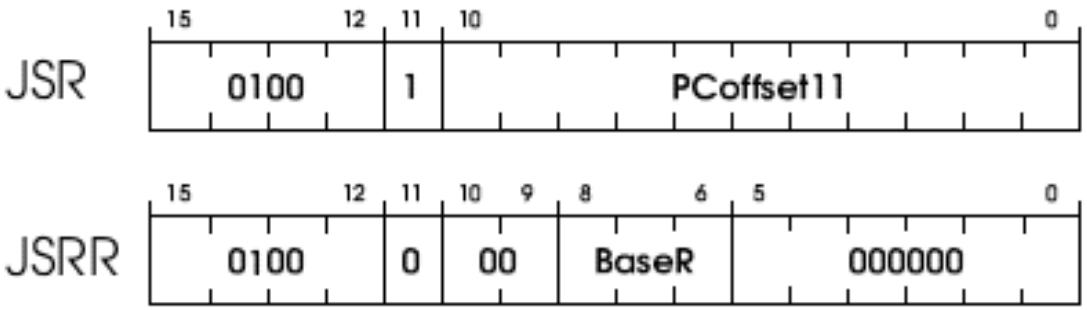


Operation

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

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

Encodings



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 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

ISA-level Tradeoffs: Instruction Length

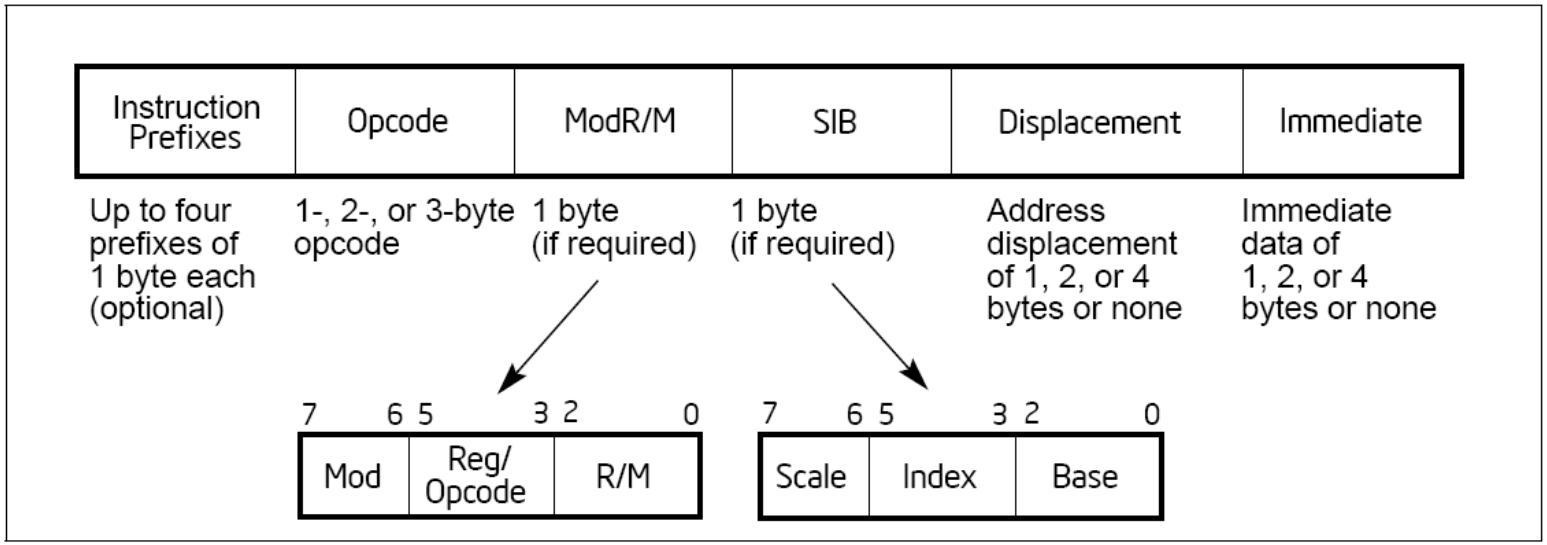
- Fixed length: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions (Why is this bad?)
  - -- Harder-to-extend ISA (how to add new instructions?)
- Variable length: Length 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

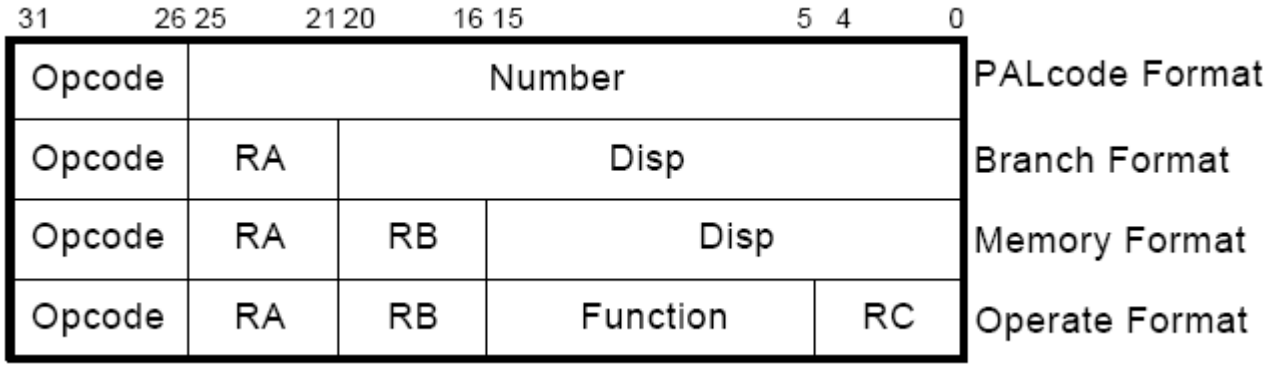
- Uniform decode: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - Ditto operand specifiers, immediate values, …
  - Many "RISC" ISAs: Alpha, MIPS, SPARC
  - + Easier decode, simpler hardware
  - + Enables parallelism: generate target address 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

x86 vs. Alpha Instruction Formats

- x86:

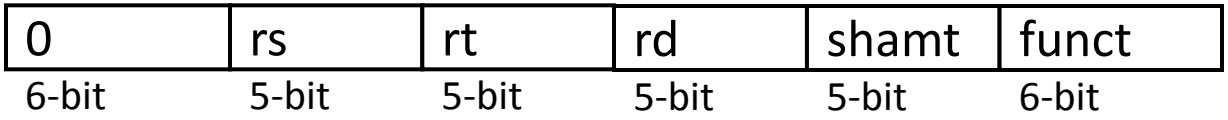


- Alpha:



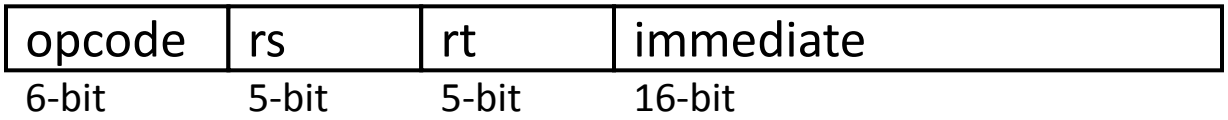
MIPS Instruction Format

- R-type, 3 register operands



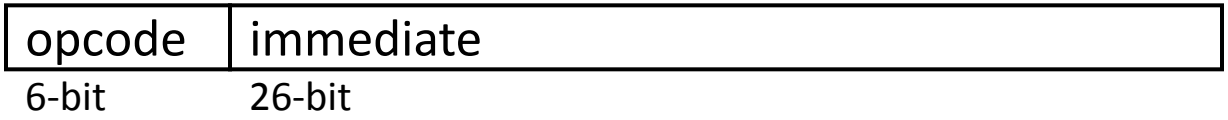
R-type

- I-type, 2 register operands and 16-bit immediate operand



I-type

- J-type, 26-bit immediate operand



J-type

- Simple Decoding
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned (2 lsb of PC must be 2b'00)
  - format and fields easy to extract in hardware

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

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-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

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:

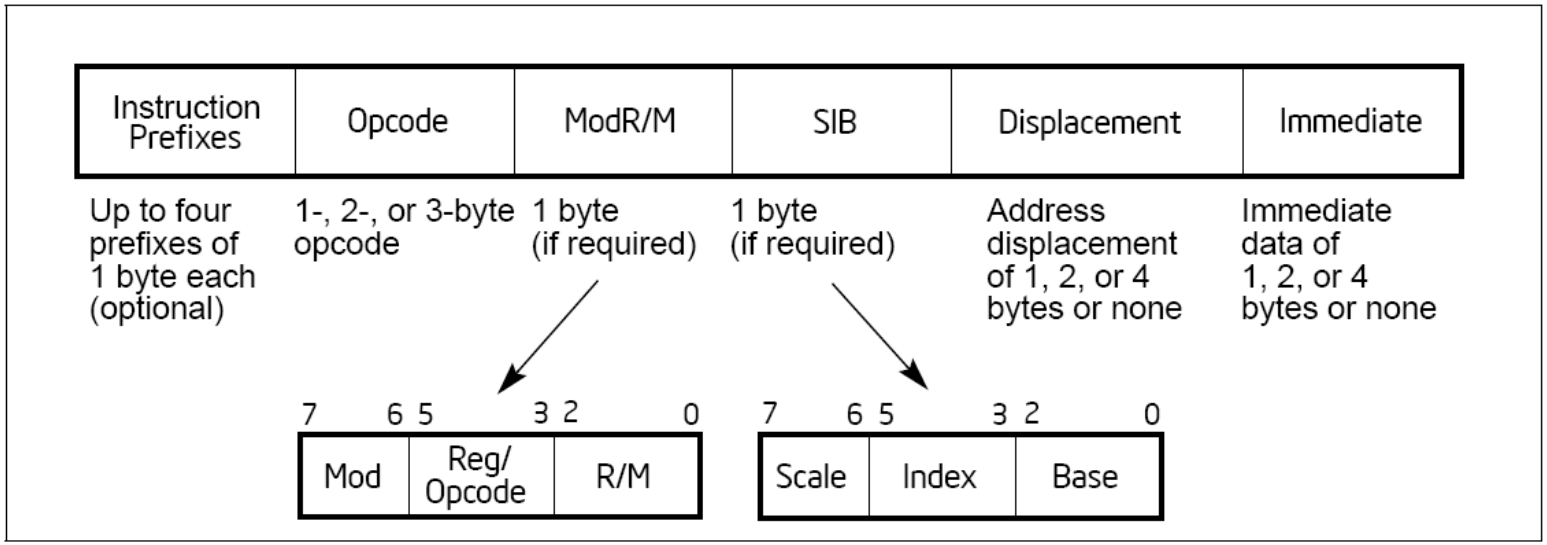
  + help better support programming constructs (arrays, pointer-based accesses)

  -- make it harder for the architect to design

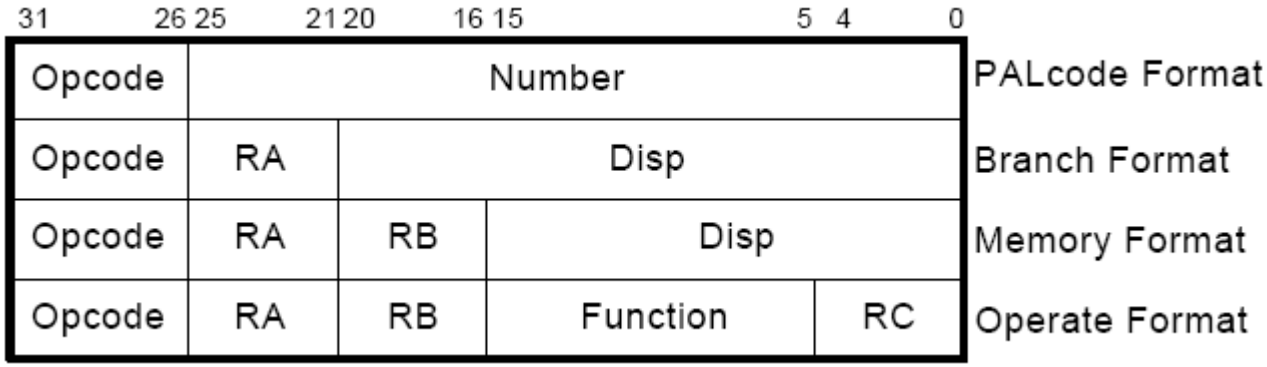
  -- too many choices for the compiler?
    - Many ways to do the same thing complicates compiler design
    - *Wulf, "Compilers and Computer Architecture," IEEE Computer 1981*

x86 vs. Alpha Instruction Formats

- x86:



- Alpha:



register indirect

absolute

register + displacement

register

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

indexed (base + index)

scaled (base + index*4)

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

NOTES:

1. 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]

X86 SIB-D Addressing Mode

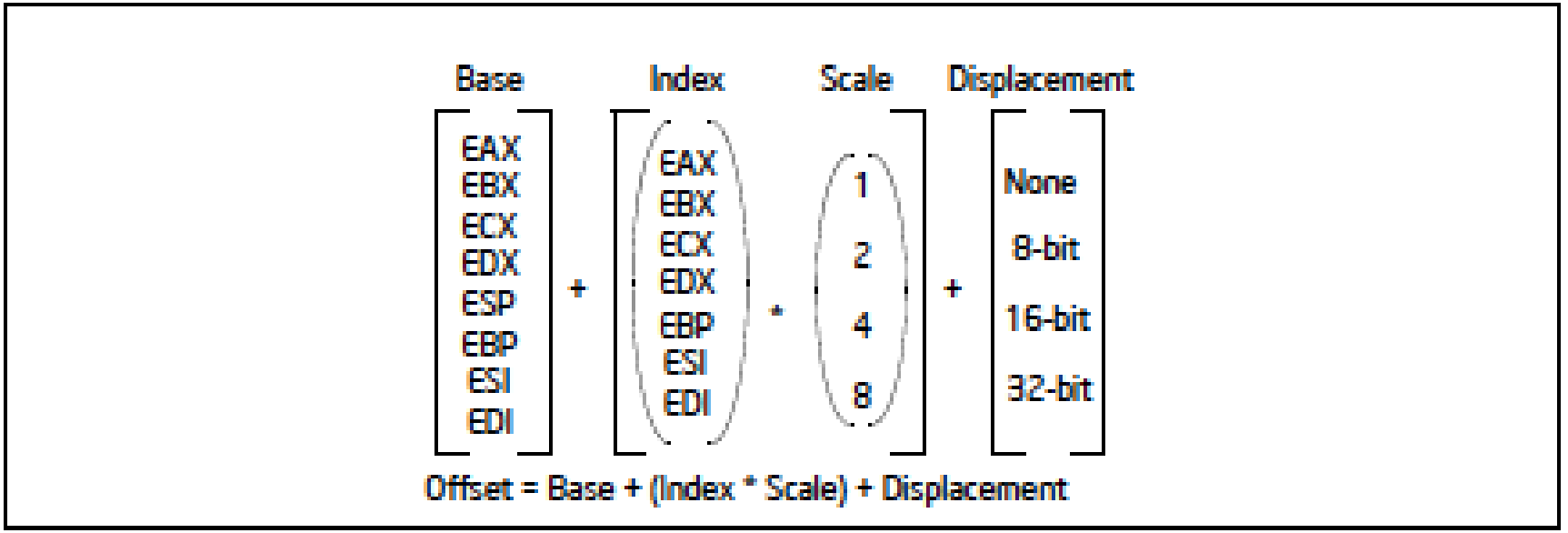


Figure 3-11. Offset (or Effective Address) Computation

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

X86 Manual: Suggested Uses of Addressing Modes

The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.
- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.
- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
  - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
  - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

  An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

X86 Manual: Suggested Uses of Addressing Modes

- **(Index * Scale) + Displacement** — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.
- **Base + Index + Displacement** — Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).
- **Base + (Index * Scale) + Displacement** — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

Other Example ISA-level Tradeoffs

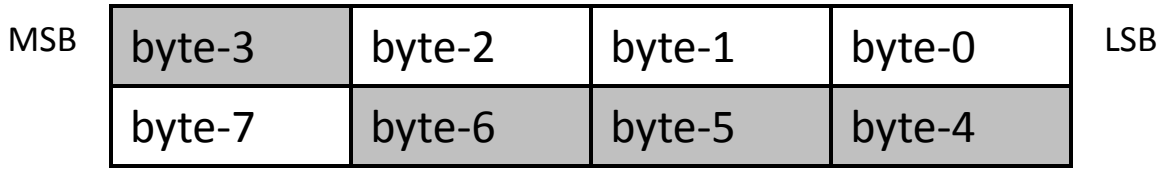
- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- …

Back to Programmer vs. (Micro)architect

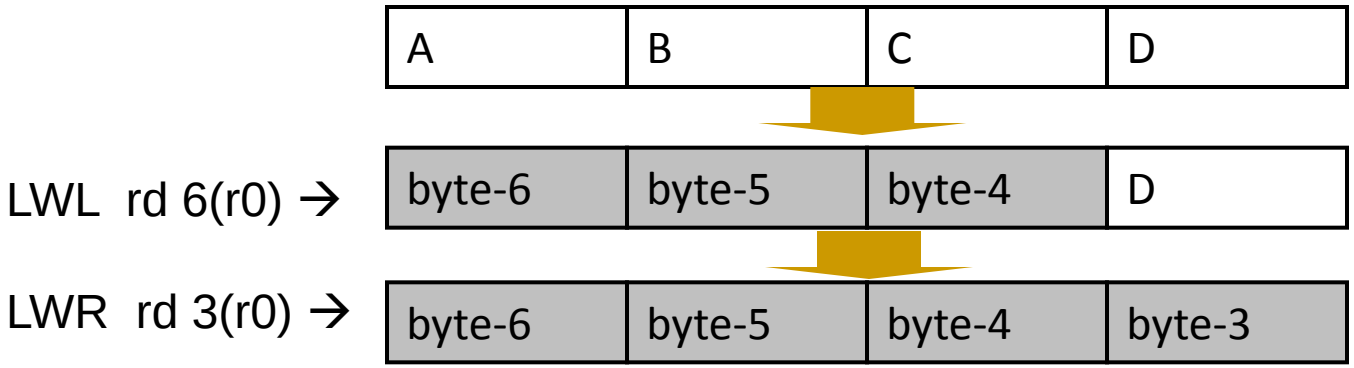
- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job

- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks fitting physical memory?
- Addressing modes
- Unaligned memory access
  - Compile/programmer needs to align data

MIPS: Aligned Access



- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the "infrequent" case



  - LWL/LWR is slower
  - Note LWL and LWR still fetch within word boundary

X86: Unaligned Access

- LD/ST instructions automatically align data that spans a "word" boundary
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location)

4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

X86: Unaligned Access

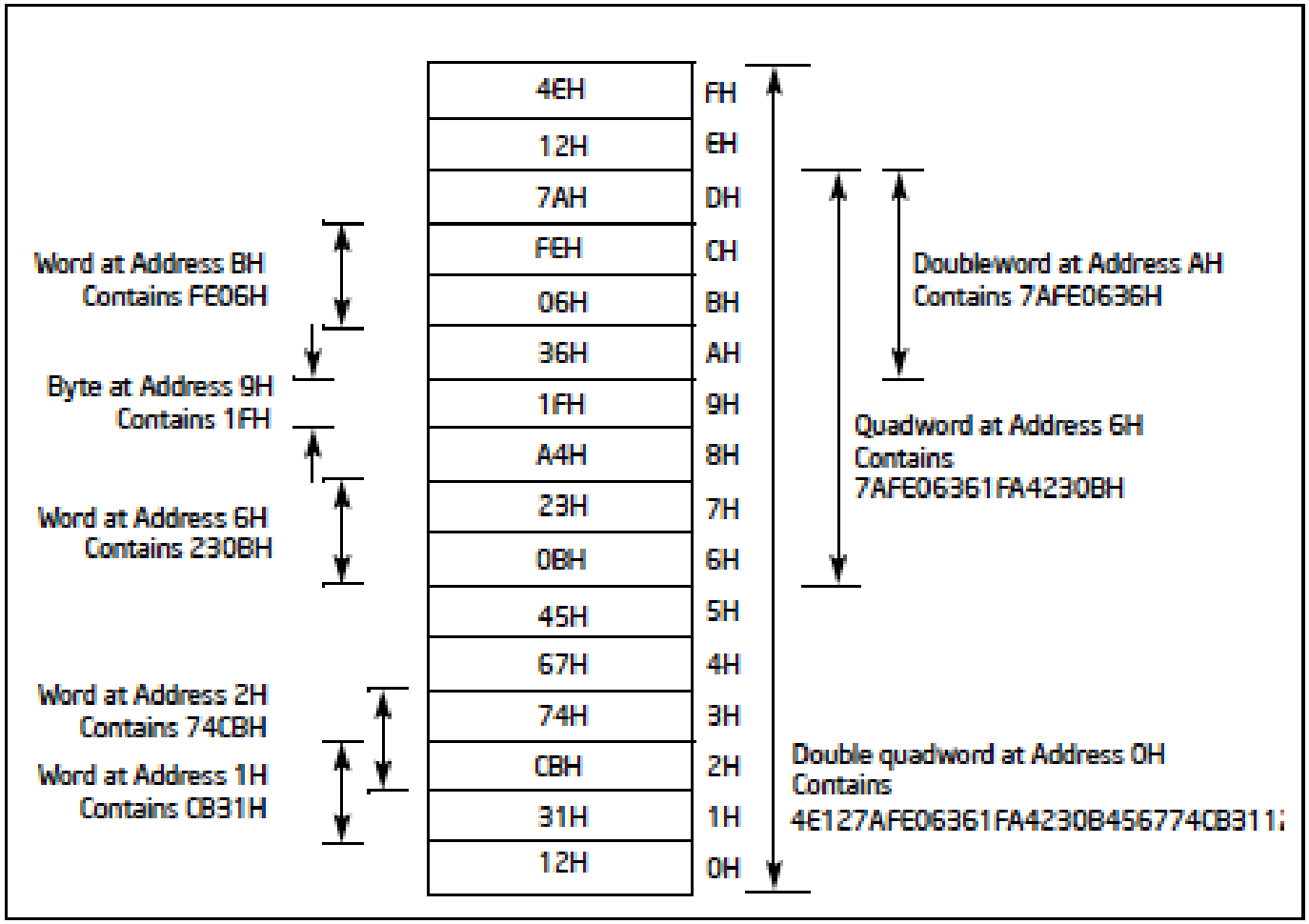


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

Aligned vs. Unaligned Access

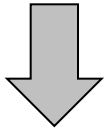
- Pros of having no restrictions on alignment
- Cons of having no restrictions on alignment
- Filling in the above: an exercise for you…

Implementing the ISA: Microarchitecture Basics

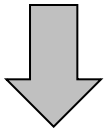
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

A = Architectural (programmer visible) state before an instruction is processed



Process instruction



A' = Architectural (programmer visible) state after an instruction is processed

- Processing an instruction: Transforming A to A' according to the ISA specification of the instruction

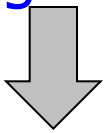
The "Process instruction" Step

- ISA specifies abstractly what A' should be, given an instruction and A
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no "intermediate states" between A and A' during instruction execution
    - One state transition per instruction
- Microarchitecture implements how A is transformed to A'
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
    - Choice 1: A → A' (transform A to A' in a single clock cycle)
    - Choice 2: A → A+MS1 → A+MS2 → A+MS3 → A' (take multiple clock cycles to transform A to A')

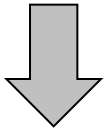
A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*

A = Architectural (programmer visible) state
at the beginning of a clock cycle



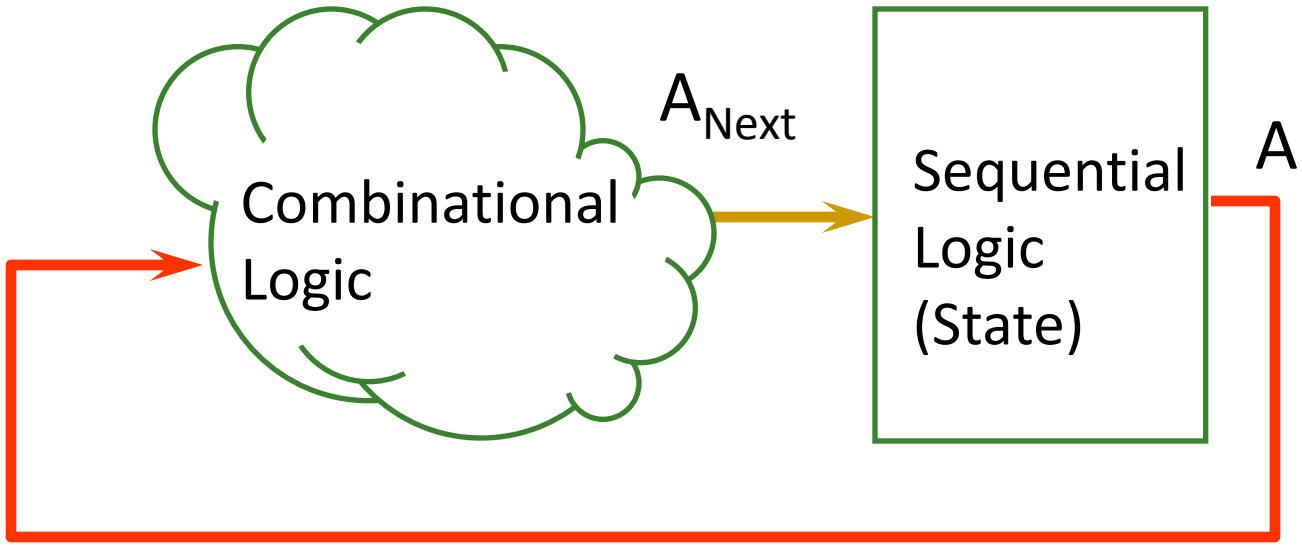
Process instruction in one clock cycle



A' = Architectural (programmer visible) state
at the end of a clock cycle

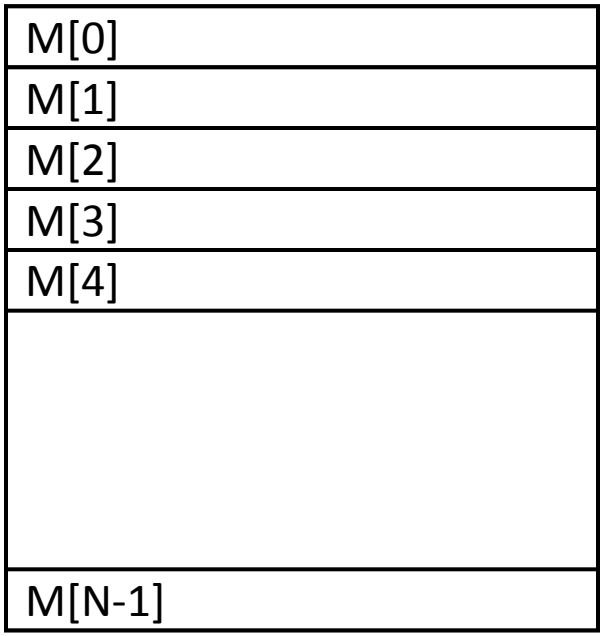
A Very Basic Instruction Processing Engine

- Single-cycle machine

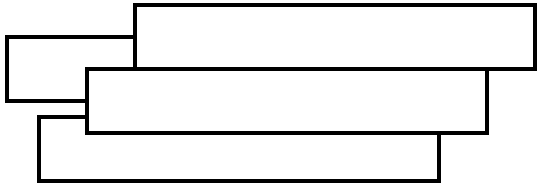


- What is the *clock cycle time* determined by?
- What is the *critical path* of the combinational logic determined by?

Remember: 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

Single-cycle vs. Multi-cycle Machines

- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines cycle time → long clock cycle time

- Multi-cycle machines
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates made only at the end of an instruction's execution
  - Advantage over single-cycle: The slowest "stage" determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

Instruction Processing "Cycle"

- Instructions are processed under the direction of a "control unit" step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six phases:

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

- Not all instructions require all six stages (see P&P Ch. 4)

Instruction Processing "Cycle" vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, each phase can take multiple clock cycles to complete

Instruction Processing Viewed Another Way

- Instructions transform Data (AS) to Data' (AS')
- This transformation is done by functional units
  - Units that "operate" on data
- These units need to be told what to do to the data

- An instruction processing engine consists of two components
  - Datapath: Consists of hardware elements that deal with and transform data signals
    - functional units that operate on data
    - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
    - storage units that store data (e.g., registers)
  - Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data

Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as data signals are operated on
  - Everything related to an instruction happens in one clock cycle

- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the previous cycle
  - Latency of control processing can be overlapped with latency of datapath operation

- We will see the difference clearly in *microprogrammed multi-cycle microarchitecture*

Many Ways of Datapath and Control Design

- There are many ways of designing the data path and control logic

- Single-cycle, multi-cycle, pipelined datapath and control
- Single-bus vs. multi-bus datapaths
  - See your homework 2 question
- Hardwired/combinational vs. microcoded/microprogrammed control
  - Control signals generated by combinational logic versus
  - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design

Flash-Forward: Performance Analysis

- Execution time of an instruction
  - {CPI} x {clock cycle time}
- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - {# of instructions} x {Average CPI} x {clock cycle time}

- Single cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

Now, we have two degrees of freedom to optimize independently