Lecture-2 (Instruction Set Architecture)
CS422-Spring 2018

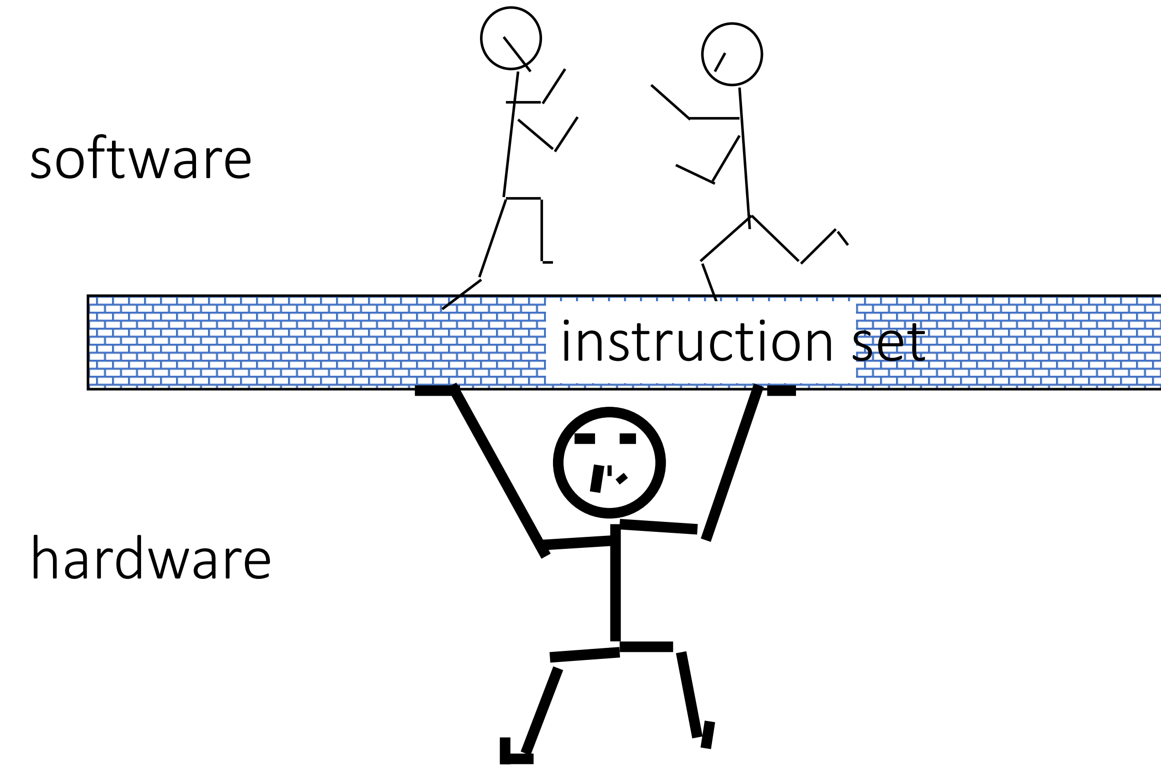
Biswa@CSE-IITK

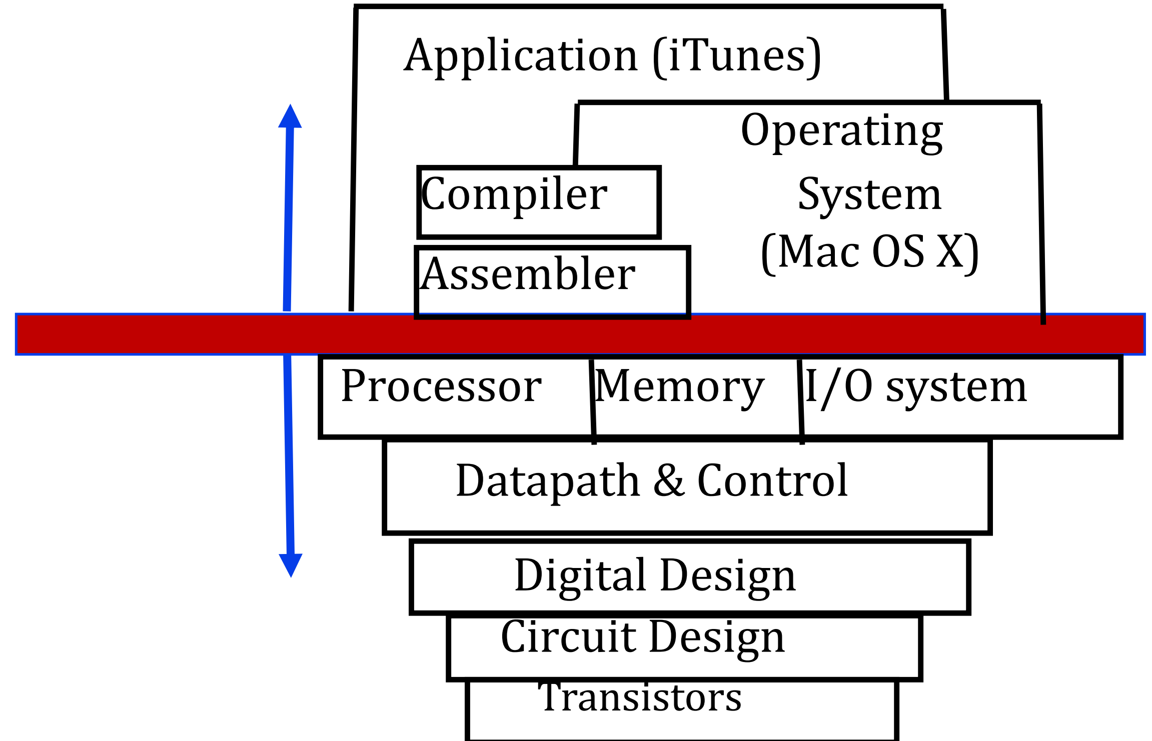
ISA

… the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.

– Amdahl, Blaaw, and Brooks, 1964

Instruction Set Architecture (ISA)





ISA

Syntax: ADD \$8 \$9 \$10 Semantics: \$8 = \$9 + \$10

Bitfield:

opcode	rs	rt	rd	shamt	funct

"R-Format"

Binary: 000000 01001 01010 01000 00000 100000

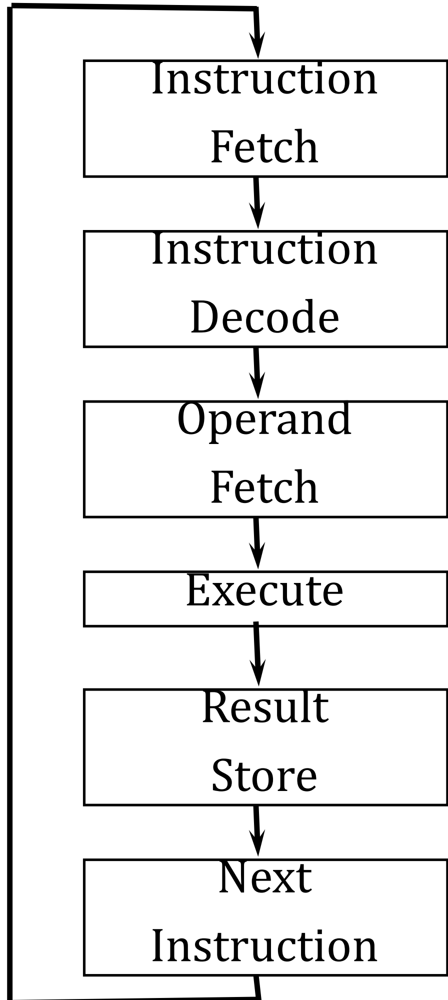
In Hexadecimal: 012A4020

Why opcode and funct?
Why shamt?
+1 point: Piazza

ISA

Syntax: ADD $8 $9 $10

Semantics: $8 = $9 + $10



Fetch next inst from memory:012A4020

opcode	rs	rt	rd	shamt	funct

Decode fields to get : ADD $8 $9 $10

"Retrieve" register values: $9 $10

Add $9 to $10

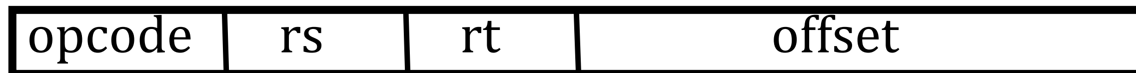
Place this sum in $8

Prepare to fetch instruction that follows the ADD in the program.

Memory Instructions: `LW $1,32($2)`

Instruction Fetch → Instruction Decode → Operand Fetch → Execute → Result Store → Next Instruction

Fetch the load inst from memory



"I-Format"

Decode fields to get : LW $1, 32($2)

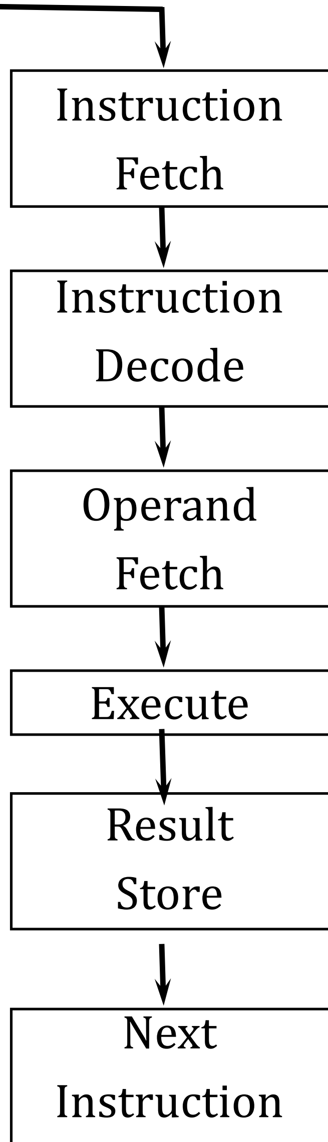
"Retrieve" register value: $2

Compute memory address: 32 + $2

Load memory address contents into: $1

Prepare to fetch instr. that follows the LW in the program.

Branch Instructions: `BEQ $1,$2,25`



Fetch branch inst from memory

"I-Format"

Decode fields to get: BEQ $1, $2, 25

"Retrieve" register values: $1, $2

Compute if we take branch: $1 == $2 ?

ALWAYS prepare to fetch instr. that follows the BEQ in the program ("delayed branch"). IF we take branch, the instr. we fetch AFTER that instruction is PC + 4 + 100.

J-type during branches

The Contract: An Architect's Contract

- To the program, it appears that instructions execute in the correct order defined by the ISA.
- As each instruction completes, the machine state (regs, mem) appears to the program to obey the ISA.
- What the machine actually does is up to the hardware designers, as long as the contract is kept.

ISA

- ISA
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture

ISA

- Basic element of the HW/SW interface

- Consists of
  - opcode: what the instruction does

  - operands: ?

Elements of an ISA

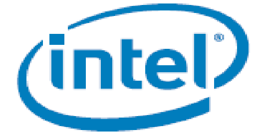
- Instruction processing style
  - Specifies the number of "operands" an instruction "operates" on and how it does so
  - 0, 1, 2, 3 address machines: 0-address: stack machine (push A, pop A, op)
    - 1-address: accumulator machine (ld A, st A, op A)
    - 2-address: 2-operand machine (one is both source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

ISA vs uarch

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for "acceleration"
  - Internals of the engine: implement "acceleration"
- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, ...
- Microarchitecture usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha) but many uarchs. *Why?*

ISA

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes
- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handling, Access Control, Priority/Privilege
- I/O: memory-mapped vs. instr, Task/thread Management, Power and Thermal Management
- Multi-threading support, Multiprocessor support



Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 1:
Basic Architecture

Uarch

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

ISA vs uarch

- ADD instruction's opcode
- Number of general purpose registers
- Number of ports to the register file
- Number of cycles to execute the MUL instruction
- Whether or not the machine employs pipelined instruction execution

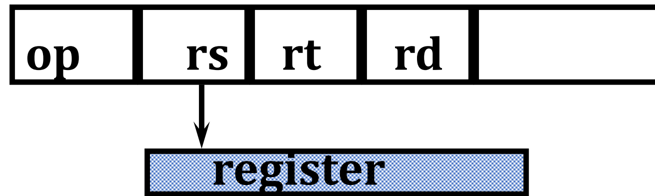
- Remember
  - Microarchitecture: Implementation of the ISA under specific design constraints and goals

Why Have Instructions

- Alternatives
  - Special purpose hardware
  - Reconfigurable hardware (FPGAs)
  - Minimize everything as much as possible?
- Simplifies interface
  - Software knows what is available
  - Hardware knows what needs to be implemented
- Abstraction protects software and hardware
  - Software can run on new machines
  - Hardware can run old software

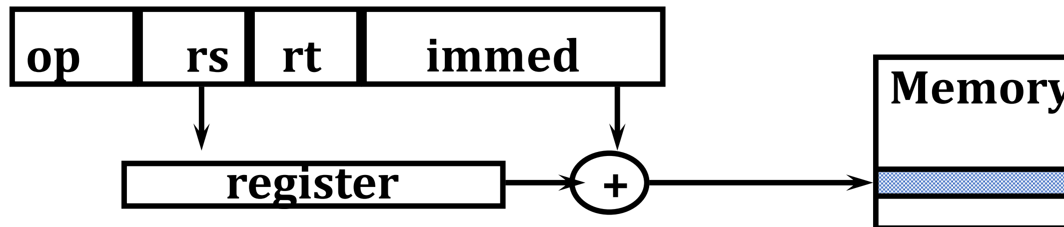
MIPS Addressing Modes

Register (direct)

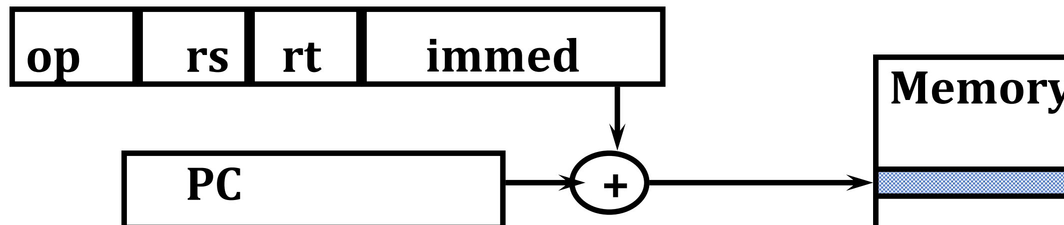


Immediate

Base+index



PC-relative



- **All instructions 32 bits wide**

How to read a 32-bit constant then?

LUI $t0 constant
ORI $t0 $t0 constant

LUI (load upper immediate) and ORI (OR immediate) instructions, will it help?

Characteristics of Good ISA

- Unambiguous
- Expressive
  - Easily describes all the algorithms that will run on this platform
- Instructions are used
  - Very complex instructions might not be used often
- (Relatively) easy to compile
- (Relatively) easy to implement well
- Implementation provides good performance, cost, etc.
- ISAs often highly reliant on microarchitecture and vice-versa
  - Some ISAs easy to implement on some microarchitectures
  - Some microarchitectures make some instructions easy to implement

Design Point

- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Considerations
  - Cost
  - Performance
  - Maximum power consumption and Energy consumption (battery life)
  - Availability
  - Reliability and Correctness
  - Time to Market
- Design point determined by the "Problem" space (application space), the intended users/*market*

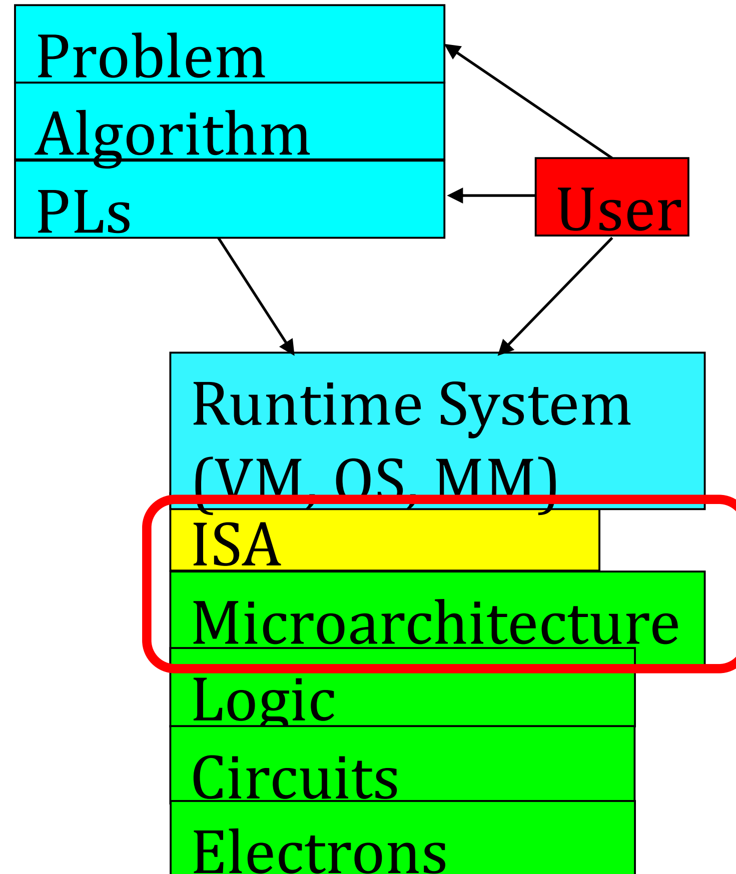
ISA Tradeoffs

- 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?*

Art ??

New demands
from the top
(Look Up)

New issues and
capabilities
at the bottom
(Look Down)



New demands and
personalities of users
(Look Up)

We do not (fully) know the future (applications, users, market) and it changes

ISA Trade-offs

- Fewer instructions
  - Pros?
  - Cons?
  - There are 1 instruction ISAs
    - subleq a, b, c ;
      - Mem[b] = Mem[b] - Mem[a] ;if (Mem[b] $\leq$ 0) goto c
- Number of registers per instruction
  - Number of bits per instruction
  - Number of registers to implement
  - Zero, One, Two, Three, Four, …
- Fixed verses Variable Length instructions

Fixed vs Variable

- Code Size: Dense (good for embedded applications?)
- Ease of decoding
- Flexibility

Control Transfer Instructions (Remind me during branches)

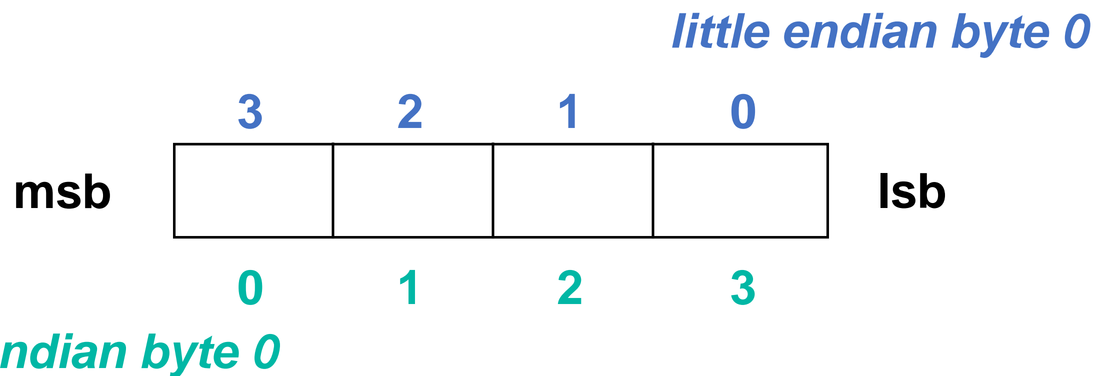
Sign Extend

- When value is sign extended, copy upper bit to full value:
  Examples of sign extending 8 bits to 16 bits:
  00001010 $\Rightarrow$ 00000000 00001010
  10001100 $\Rightarrow$ 11111111 10001100
- When is an immediate operand sign extended?
  - Arithmetic instructions (add, sub, etc.) always sign extend immediates *even for the unsigned versions of the instructions!*
  - Logical instructions do not sign extend immediates (They are zero extended)
  - Load/Store address computations always sign extend immediates
- Multiply/Divide have no immediate operands however:
  - "unsigned" $\Rightarrow$ treat operands as unsigned
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended

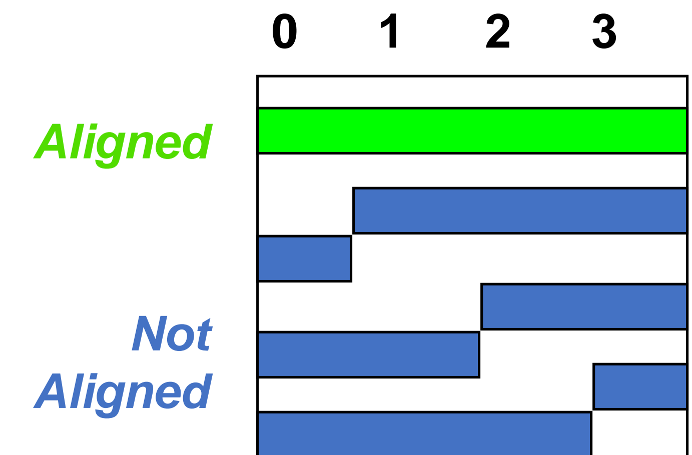
Endianness and Alignment (*Pointers)

- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

Think about a network packet from BE to LE machine

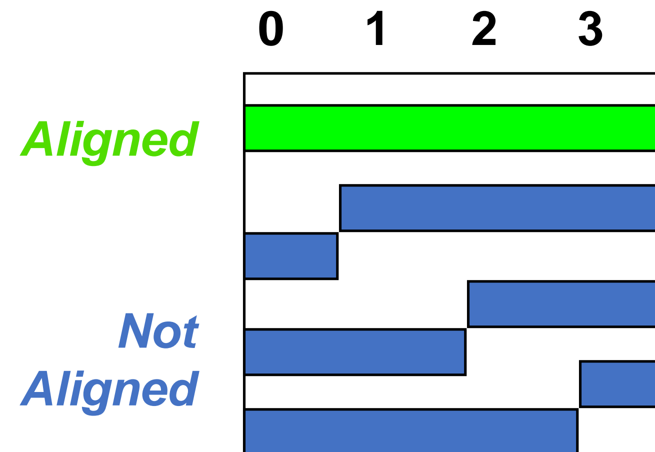


Alignment:
require that objects fall on address that is multiple of their size.



Alignment

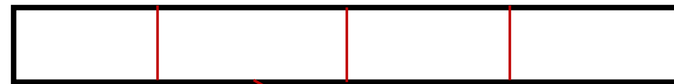
- Object of size s bytes at byte add. A is aligned if A mod s = 0
- Alignment for faster transfer of data ?
- Why fast ??
- Think about memory.

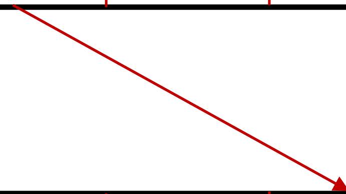


Bit More

- LB 0x1, r1

4-byte chunk





Register r1

Hang on? What determines this 32-bit & 64-bit ?

- Memory Size
- Register Size
- OS
- Limited by Physical and virtual address width
- Instruction Size

Hang on? What determines this 32-bit & 64-bit ?

- Memory Size
- Register Size
- OS
- Limited by Physical and virtual address width
- Instruction Size

More Questions

- 32-bit OS on 32-bit and 64-bit ISA
- What about other apps?
- Old legacy code ?
- Test it
- Write a simple program
- Compile using –m32 with gcc and then disassemble using gdb
- Try playing with the pointers

Orthogonality

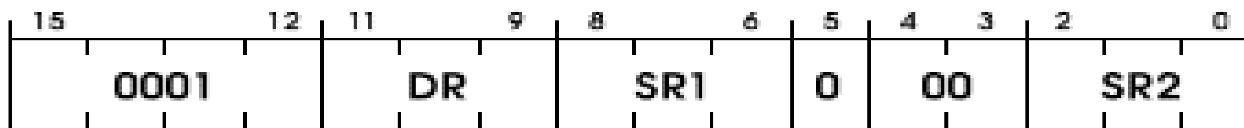
- Orthogonal ISA:
  - All addressing modes can be used with all instruction types
  - Example: VAX
    - (~13 addressing modes) x (>300 opcodes) x (integer and FP formats)

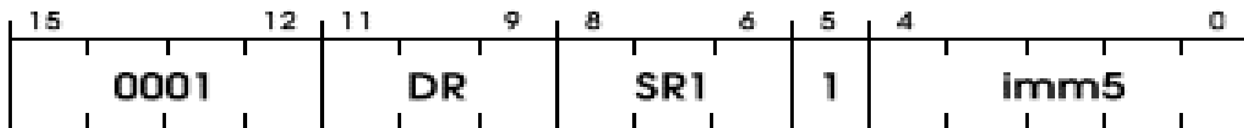
- Who is this good for?

- Who is this bad for?

Orthogonality

Encodings





Operation

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

Simple vs Complex

- 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

Simple vs Complex

- 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

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)

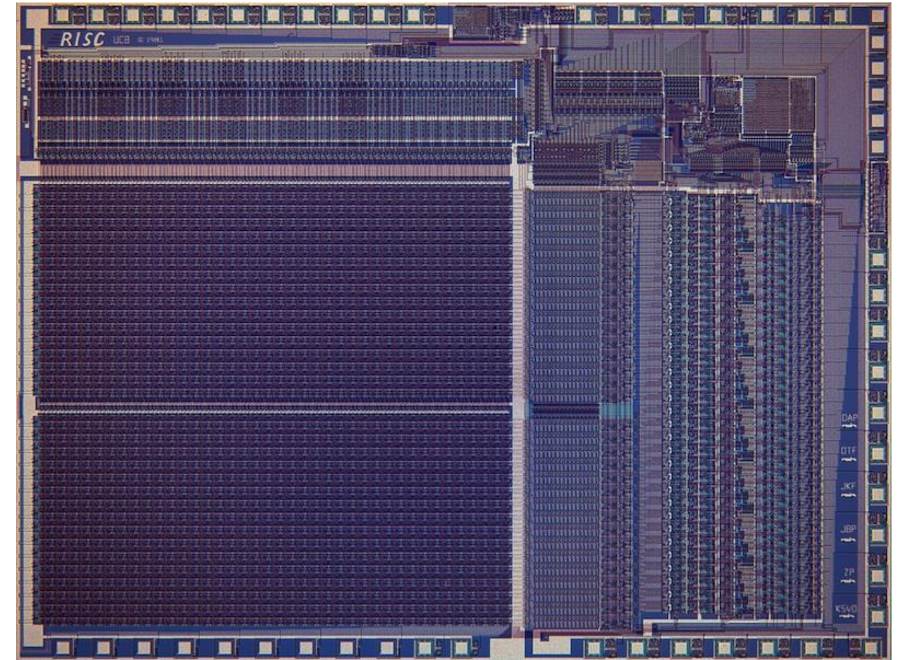
Top 10 x86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

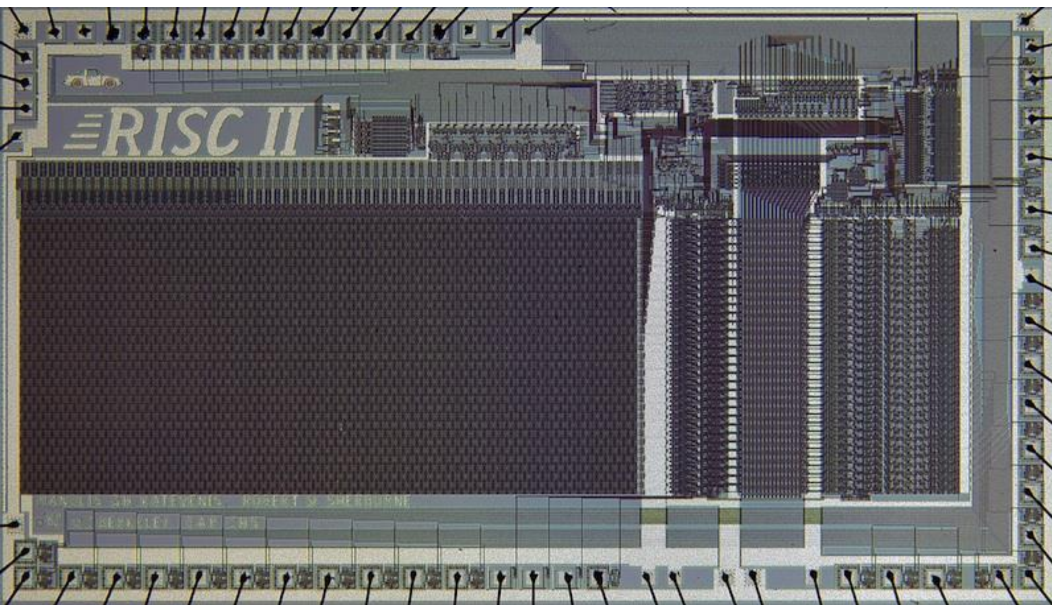
- Simple instructions dominate instruction frequency

RISC vs CISC

RISC-I (1982) Contains 44,420 transistors, fabbed in 5 µm NMOS, with a die area of 77 mm^2, ran at 1 MHz. This chip is probably the first VLSI RISC.



Berkeley RISC Chips



RISC-II (1983) contains 40,760 transistors, was fabbed in 3 µm NMOS, ran at 3 MHz, and the size is 60 mm^2. Stanford built some too…

CISC vs RISC

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

ARM Instructions (Thumb and Zazzle)

Compressed 16-bit instructions ~ 32-bit ones

De-compressed before decode and execution

Typically 60 to 65% of the size of the normal ARM code

Embedded applications that use ROM

For rich features – switch to ARM mode from thumb mode

ISA In a Nutshell

- Which architecture to use: Stack, Accumulator, R-R, R-M
- #Registers
- Trade-offs in terms of #operands, #registers, ….
- Big vs Little Endian
- Importance of Alignment
- Which instructions are dominant and which addressing modes are dominant ?
- Fixed vs Variable Encoding
- RISC vs CISC
- Pseudo - instructions
- ISA vs micro-architecture

BACK-UP SLIDES

BACK-UP SLIDES

- Reading a constant

0000 0000 0011 1101 0000 1001 0000 0000

- lui $t0, 61

Then

Ori $t0, $t0, 2304

Confused?

Welcome to the world of pseudo instructions

li $t0, 32-bit constant

MIPS in Nutshell

- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement, no indirection
- **16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =,≠, no integer condition codes
- **Delayed branch**
  - execute instruction after a branch (or jump) even if the branch is taken (Compiler can fill a delayed branch with useful work about 50% of the time)

Different ISAs

- 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

ISA: What Else?

- 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