Instruction Set Architecture
ISA

Today's topics:

- Note: desperate attempt to get back on schedule
  - we won't cover all of these slides – use for reference
- Risk vs. CISC
- x86 does both
- ISA influence on performance & complexity
- some basic examples
- fetch and decode issues

ISA

- What is it really?
  - set of instructions
  - THE HW/SW contract
    - » compiler correctly translates source code to the ISA
    - » assembler translates to relocatable binary
    - » linker solidifies relocatables into object code
    - » HW promises to do what the object code says
  - upside
    - » ISA provides "reasonable" SW abstraction of the HW
    - » what is missing?
  - downside
    - » reverse compatible requirement → "hide what you can" effect
- Options
  - fixed vs. variable length, instructions (RISC, CISC), memory modes, etc.

Instruction Characteristics

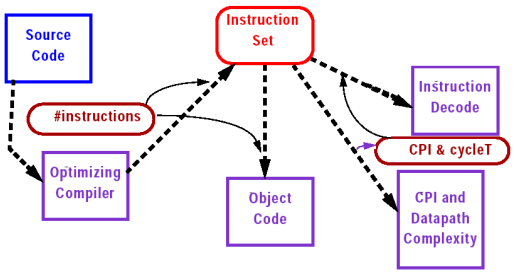
- Simple operation
  - op-code
- Operand addressing
  - explicit – source address is explicit
  - implicit – source address implied by the op code or architecture
- Address target
  - memory (CISC) vs. register (RISC)
  - RISC exception: load and store, jumps and calls
- # of operands – 0, 1, 2, 3
  - 0 → stack machine: pop 0, 1, or 2 then push result
  - 1 → single accumulator: acc ← acc OP address target
  - 2 → GPR machine: R[RS0] ← R[RS0] OP R[RS1]
  - 3 → GPR machine: R[RS0] ← R[RS1] OP R[RS2]

What Instructions are Needed

- Very few if you want to get bonkers
  - PDP-0 had a 3-bit opcode field – what 8 would you pick?
    - » hint: 1 was HALT
  - Ivan's 1 instruction computer only used MOVE
    - » saves op-code bits since there's only 1 and you don't need to specify it explicitly
- More normal – varies significantly with segment
  - arithmetic and logical
    - » choice of what data types to support
    - » fused: MAC
  - control: branch, jump, call, return, branch
  - OS – ignore these for now
  - string
  - bit field manipulation

ISA Affects Everything

The problem: XEQ_t = #Instructions x CPI x cycle-time
Iset influences everything + legacy effect!



Classifying ISA's

Based on CPU internal storage options
AND operand-arity

Operand Storage in CPU	Where are they other than memory
Number of explicit operands named per instruction	How many? Min, Max - maybe even average
Addressing Modes	How is the effective address for an operand calculated? Can all operands use any mode?
Operations	What are the options for the opcode?
Type and size of operands	How is typing done? How is the size specified

These choices critically affect - #instructions, CPI, and cycle time

Form and Function are Related

Consider the class pro's and con's

Machine Type	Advantages	Disadvantages
Stack	Simple effective address Short instructions Good code density Simple I-decode	Lack of random access. Efficient code is difficult to generate. Stack is often a bottleneck.
Accumulator	Minimal internal state Fast context switch Short instructions Simple I-decode	Very high memory traffic
Register	Lots of code generation options. Efficient code since compiler has numerous useful options.	Longer instructions. Possibly complex effective address generation. Size and structure of register set has many options.

Modern Choice - GPR

- **Why?**
  - **IBM legacy to some extent – they were dominant at the right time**
  - **compiler optimizations for GPR**
    - » **simpler cost model so easier to evaluate options**
    - » **register scheduling easier than memory operations**
    - » **stack lost due to compilers**
      - **and JB who came from IBM to be CEO of Burroughs**
      - **the company went down the tubes in 3 years**
      - **not clear that stack machines deserved the bad rap they got in history**
- **Platform independence**
  - **If GPR's dominate then it's a bigger pain for the compilers to also handle something that is very different**
  - **software lives forever and HW evolves very quickly**
- **Compiler technology is still key**
  - **to extracting the performance of the HW**
  - **advanced today for the GPR world**

Sample Comparison

- Examine datapath and control strategies
- Datapath assumptions for this example
  - only direct addressing
  - 8 bit opcode
  - 16 bit registers
  - 16 bit memory address field
  - no byte or half-word to keep things simple
    - » use 32-bit values
  - simple tri-state bus as well
- Control assumptions
  - micro-code like here
  - In reality implemented by FSM controller

Instruction Formats

Accumulator - 24 bit instruction

8-bit OpCode	16-bit Memory Address

Stack - also 24 bit but most instructions will be 8-bit (pack-em)

8-bit OpCode	16-bit Memory Address

Register - 2 explicit operands (3 explicit is obvious) - 28 bits

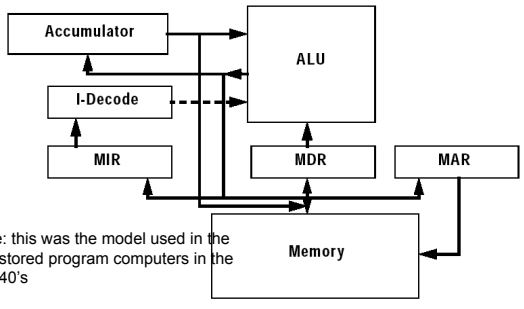
8-bit OpCode	4-bit Reg-addr	16-bit Memory Address

- could pack but instruction word allignment would be a problem.

Things to note

- Abbreviations
  - IR – instruction register
  - MAR – memory address register
  - MDR – memory data register
  - ALU – arithmetic and logical unit
- Ridiculously simple example
  - Ignores many critical issues
  - Idea is to convey what gets built
    - » and how to start thinking about an implementation

Accumulator Datapath



Note: this was the model used in the first stored program computers in the late 40's

Accumulator Control

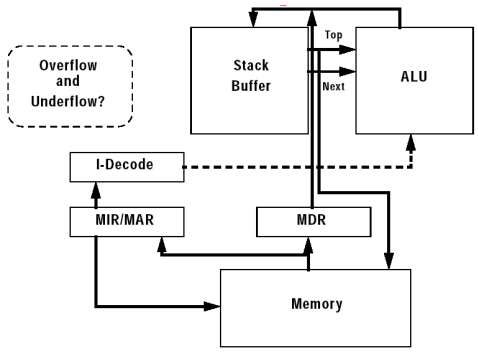
Loads
- Read Memory, Enable Memory to Accumulator
- Load Accumulator

Stores
- Enable Accumulator to MBUS
- Write Memory

ALU Op's
- Read Memory
- Enable ALU to Accumulator
- Load Accumulator

Branch - just like an IFetch but with PC as address source
- Read Memory
- Enable Memory to MIR
- Load MIR

Stack Datapath



Stack Control (over simplified)

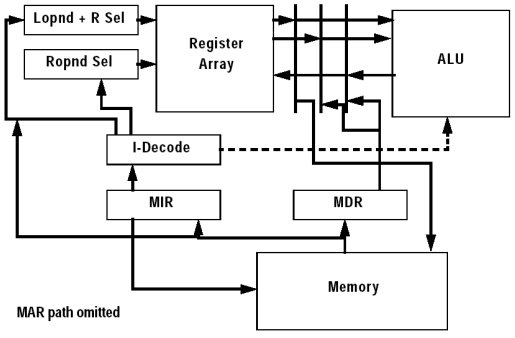
Loads
- Read Memory
- Push

Stores
- Enable Top to MBUS, Write Memory
- Pop

ALU Op's
- Load Top or Next
- Pop or not

Branch - just like an IFetch but with PC as address source
- Read Memory
- Enable Memory to MIR
- Load MIR

GPR Datapath



GPR Control

- Loads - just like accumulator but select Reg.
- Stores - just like accumulator but select Reg.
- Branch - same as all the rest
- ALU OPs - whoa!
  - select Left operand and result register
  - decide whether you want memory or a register for the right operand
  - note minimum of 3 busses between the Register array and the ALU
- Effective memory address calculation
  - immediate
  - register pointer
  - register alu-op registers
    - possible auto-increment or decrement on one of them

Text's classification for ISA types

- **(# of memory operands, Max ALU operands)**

# Memory Ops per typical ALU instruction	Max ALU operands allowed	Examples
0	2 3	IBM RT-PC SPARC, MIPS, HP-PA, PowerPC, ALPHA
1	2 3	PDP-10, M6800, IBM 360, Intel 90x86 IBM 360RS
2	2 3	PDP-11, National 32x32, IBM 360SS, VAX NEC S1
3	3	VAX - blech!

(0,3) Reg-Reg: Pro's and Con's

- **Pure RISC**
  - **only load and store go to memory**
- **Advantages:**
  - **simple fixed length instruction**
    - » **simplifies decode**
  - **simple code generation**
  - **simple cost model**
    - » **since CPI for instructions will be known**
    - » **exception is load store**
      - **and in today's high frequency world some things are a little more iffy**
- **Disadvantages**
  - **high IC → Imem footprint**
  - **some instructions don't need all of the instruction word bits**
    - » **→ mem footprint**

(1,2)/(1,3) Reg-Mem P's & C's

- **Evolved RISC and old CISC – go figure?**
  - **some new RISC machines**
    - » **speculative loads**
    - » **predicated or deferred loads**
- **Pro's**
  - **no need to do a load before a use**
  - **instruction format is still simple**
  - **improved code density**
- **Con's**
  - **source operands are not equivalent in (1,2)**
    - » **1 reg source value is destroyed with result value**
    - » **memory address field needs to be bigger than register field**
    - » **CPI varies for anything from memory: cache, main, disk??**

(3,3) mem-mem P's and C's

- Ultimate gaggy CISC
  - extinct now and likely to remain that way
- Pro's
  - small instruction footprint?
    - » not clear given need for 3 large addresses
  - doesn't waste a register for touch once data
    - » register file consumes a lot of power → heat
- Con's
  - large variation in instruction size
  - large variation in CPI
    - » compiler just gives up
  - high memory pressure
    - » memory is always the bottleneck
  - slowest machine imaginable

Memory Addressing

- Natural questions
  - what is accessed: byte, word, multiple words??
    - » legacy today is byte addressing which is silly
  - disks, main memory, caches, and the memory bus
    - » all organized with some "chunk" size in mind
      - caches have chunk = line
      - memory & bus chunk matches lowest level cache line size
      - disks deliver in page sized chunks
- Alignment problems are possible
  - accessing a word or double which crosses a cache line boundary
    - » requires 2 references rather than 1
    - » more CPI ambiguity
    - » bad idea but guess who allows this?

Words and byte order

- The Lilliputian Wars
  - IEEE Computer article by Prof. James Finnegan
    - » Oceanview Univ, Oceanview, KS
- Big vs. Little Endian
  - Big Endian – byte 0 is the MSB
  - Lil' Endian – byte 0 is the LSB
- Is this a problem?

Words and byte order

- The Lilliputian Wars
  - IEEE Computer article by Prof. James Finnegan
    - » Oceanview Univ, Oceanview, KS
- Big vs. Little Endian
  - Big Endian – byte 0 is the MSB
  - Lil' Endian – byte 0 is the LSB
- Is this a problem?
  - yes – I/O delivers bytes in numerical order
- Today's solution
  - an Endian bit in a control register
  - determines which side of words fill first (MSB vs. LSB)

Processor Alignment Checks

- Common convention
  - expect aligned data
  - opcode determines what you load or store
    - » LDB – byte; LDW – word; etc.
  - NOTE:
    - » we're in 64-bit processor land now but we define word = 32b
- Hardware checks for valid byte address based on load or store type
  - byte – any address is legal
  - half word – address must have a low order bit = 0 else trap
  - word – addr must have 2 low order bits = 0 else trap
  - double – addr. must have 3 low order bits = 0 else trap

Typical Address Modes I

Mode	Example Instruction	Meaning	Use
Register	Add R4, R3	Regs[R4] <- Regs[R4] + Regs[R3]	All RISC ALU operations
Immediate	Add R4, #3	Regs[R4] <- Regs[R4] + 3	for small constants - problems?
Displacement	Add R4, 100(R1)	Regs[R4] <- Regs[R4] + Mem[100 + Regs[R1]]	accessing local variables
Register deferred or Indirect	Add R4, (R1)	Regs[R4] <- Regs[R4] + Mem[Regs[R1]]	pointers
Indexed	Add R3, (R1 + R2)	Regs[R3] <- Regs [R3] + Mem[Regs[R1] + Regs[R2]]	array access - R1 is the base, R2 is the index
Direct or absolute	Add R1, (1001)	Regs[R1] <- Regs[R1] + Mem[1001]	problems?

Typical Address Modes II

Mode	Example Instruction	Meaning	Use
Memory Indirect or Memory Deferred	Add R1, @R3	Regs[R1] <- Regs[R1] + Mem[Mem[Regs[3]]]	If R3 holds a pointer address, then result is the full dereferenced pointer
Autoincrement in this case post increment note symmetry with autodec	Add R1, (R2) +	Regs[R1] <- Regs[R1] + Mem[Regs[R2]]; Regs[R2] <- Regs[R2] + d	Array walks - if element of size d is accessed then pointer increments auto
Autodecrement in this case predecrement	Add R1, - (R2)	Regs[R2] <- Regs[R2] - d; Regs[R1] <- Regs[R1] + Mem[Regs[R2]];	array walks, with autoinc useful for stack implementation
Scaled	Add R1, 100 (R2) [R3]	Regs[R1] <- Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]	array access - may be applied to indexed addressing in some machines

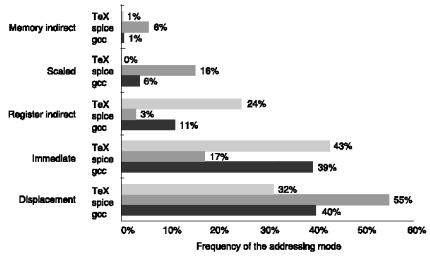
d ::= size of an element

Mode Mind Games

- Best way to understand utility of addr. modes
  - pick a few small loops from your own codes
  - see what instructions would be required using various modes
    - » e.g. if you don't have a mode then effective address will need to require extra instructions in your object code
  - think about how you would encode the instruction set that contains what you like
  - do a block diagram of the effective address path that would support your instruction set.
    - » often an integer word add is a good measure of what can be done in 1 clock cycle.
    - » estimate how many cycles each address mode would require
- Hint
  - questions like this tend to show up on the first mid-term

Mode Importance via benchmarks

based on a VAX which supported everything - SPEC89 codes

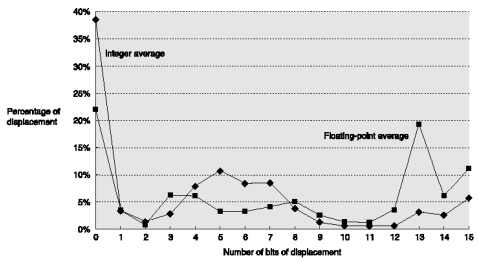


Address Field Size?

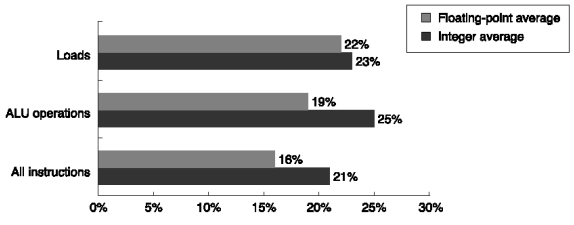
- Measure and optimize for the common case
  - Analyze your programs
    - » get dynamic instruction traces or counts
    - » want a broad benchmark spectrum & optimized compiler runs
- Choose
  - displacement field size
  - immediate or literal size
  - address modes
  - register file size
- Then evaluate cost implications
  - datapath → CPI and cycle time
  - code density and instruction decoding overhead
  - ISA encoding overhead

Displacement Values

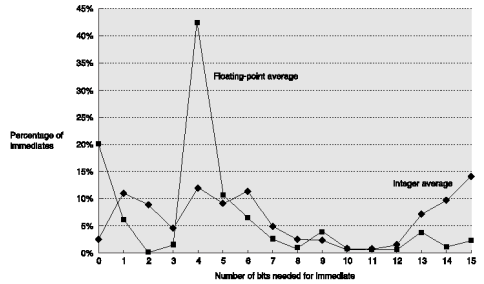
SPEC2000 based



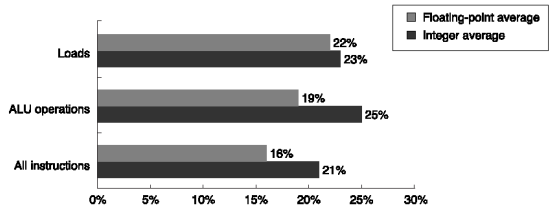
Do we need Immediate data?



OK – what size immediate



SPEC2000 Operand Sizes



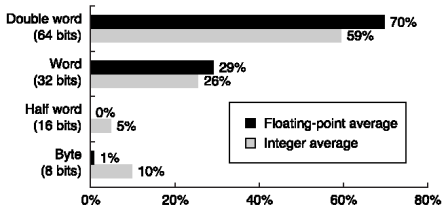
DSP Address Modes

- Data is typically an infinite stream
  - hence model memory as a circular buffer
    - » register holds a pointer to current access
    - » 2 registers hold start and end points
    - » auto increment/decrement + end detection
  - modulo or circular mode
- FFT is a common app.
  - butterfly or shuffle is the common access stride
  - bit-reverse mode
    - » reverses n low order bits in the address
    - » n is a parameter since it varies with FFT step
- Importance: 54 DSP codes on a TI C54x DSP proc.
  - immediate, displacement , reg. indirect, direct = 70%
  - auto inc/dec = 20%
  - all other modes collectively = 10%

Media and Signal Processing

- New data types
  - vertex
    - » 4 float vector: x, y, z, w
  - pixel
    - » 4 byte sized int's: R, G, B, A (transparency)
- New numeric types
  - fixed point numbers between -1 and 1
  - all mantissa: fixed point between 0 and 1
- New operations
  - inner product is very common
    - » fused instructions = MAC
    - » usage: b = ax + previous b

The Ubiquitous x86



Summary

- Simple is good
  - compilers → better code generation and optimization quality
  - machine → speed
- Beware the 90-10 rule though
  - 10% of the static instructions take 90% of the time
    - » must use dynamic counts/traces
- Can we punt on complex instructions?
  - depends on performance
    - » the 10% can get arbitrarily bad
  - depends on cost
    - » some new types, modes, etc. are almost free
  - or sadly
    - » some idiot just wanted to fingerprint the design