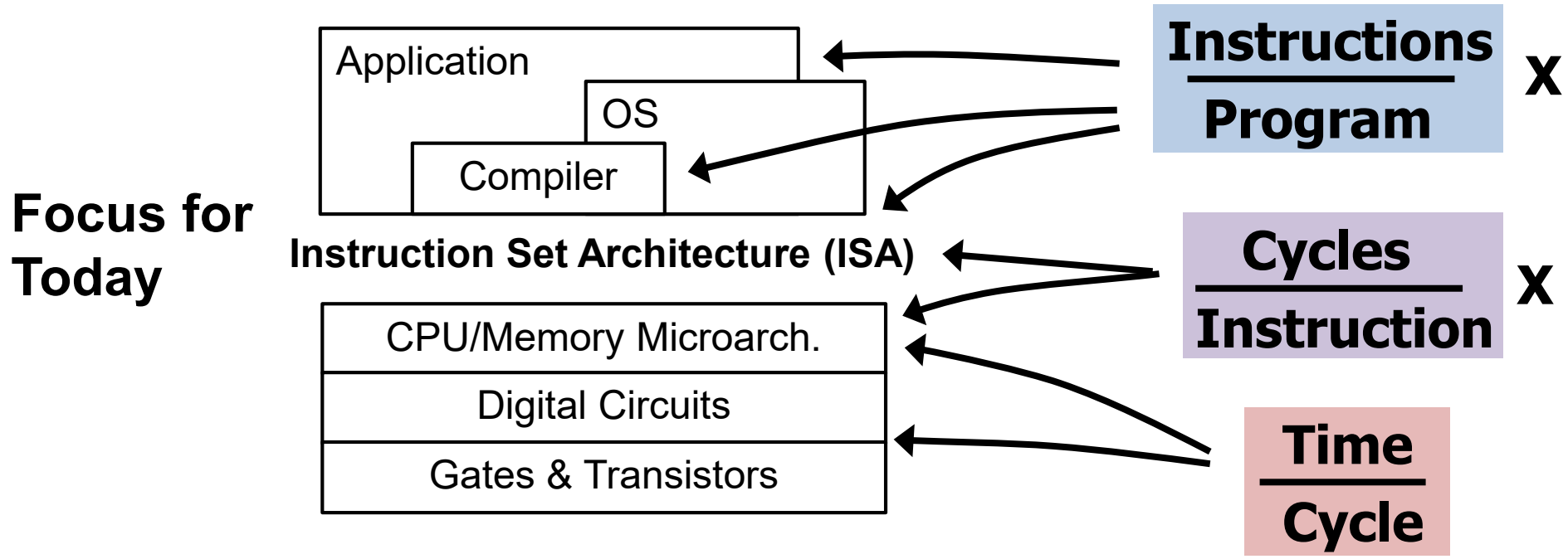
UCLA CS 251a: Advanced Computer Architecture

Prof. Tony Nowatzki

Hardware/Software Interfaces

Slide History/Attribution Diagram:

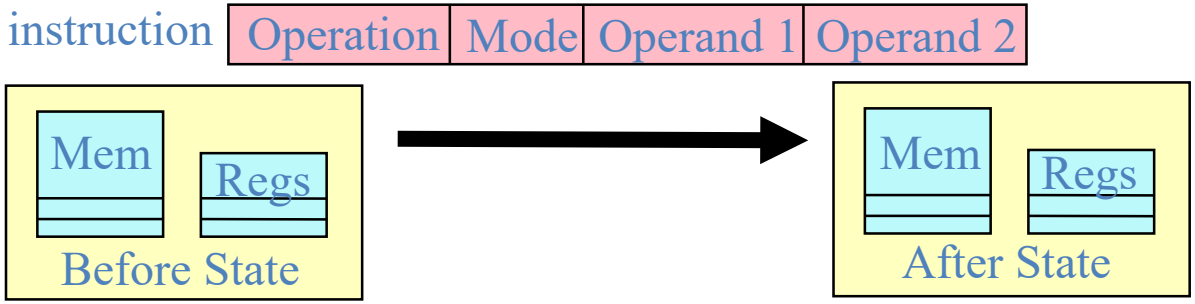
Computer System Layers



Basic Concepts / Review

What Is An ISA?

- The **"contract"** between software and hardware
  - **Functional definition** of operations, modes, and storage locations supported by hardware
  - **Precise description** of how to invoke, and access them
- Instruction Concept



- Execution Model: When to perform each instruction
- ISAs can hide some details about hardware choices:
  - How operations are implemented
  - Which operations are fast and which are slow and when
  - Which operations take more power and which take less

A Language Analogy for ISAs

- A ISA is analogous to a human language
  - Allows communication
    - Language: person to person
    - ISA: hardware to software
  - Many common aspects
    - Part of speech: verbs, nouns, adjectives, adverbs, etc.
    - Common operations: calculation, control/branch, memory
  - Many different languages/ISAs, mostly they are similar…
    - Sometimes fundamental differences
  - Both evolve over time
- Key differences: ISAs must be unambiguous
  - ISAs are explicitly engineered and extended

Compilers and Assembly

```
int array[100], sum;
void array_sum() {
   for (int i=0; i<100;i++) {
      sum += array[i];
   }
}
```

```
array_sum_loop:
    lw $3, 0($1)   #array
    lw $4, 0($2)   #sum
    add $4, $3, $4
    sw $4, 0($2)
    addi $1, $1, 1
    addi $5, $5, 1
    li $6, 100
    blt $5, $6, array_sum_loop
```

- **Compiler**: translates program to **assembly**
  - Parsing and straight-forward translation
  - Code optimization (using abstract representation, e.g. inf. Registers)
  - Backend code generation (select registers, arrange instructions, etc.)

Assembly & Machine Language

- **Assembly language**
  - Human-readable representation
- **Machine language**
  - Machine-readable representation
  - 1s and 0s (often displayed in "hex")
- **Assembler**
  - Translates assembly to machine

Machine code	Assembly code
x9A00	CONST R5, #0
x9200	CONST R1, array
xD320	HICONST R1, array
x9464	CONST R2, sum
xD520	HICONST R2, sum
x6640	LDR R3, R1, #0
x6880	LDR R4, R2, #0
x18C4	ADD R4, R3, R4
x7880	STR R4, R2, #0
x1261	ADD R1, R1, #1
x1BA1	ADD R5, R5, #1
x2B64	CMPI R5, #100
x03F8	BRn array_sum_loop

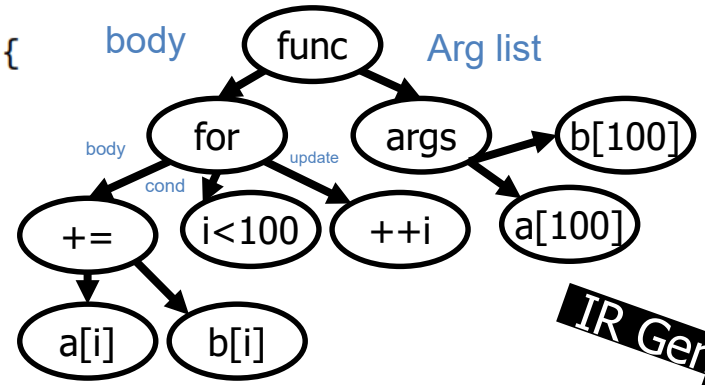
Compiler Overview

Original Program

```
int func(int a[100], int b[100]) {
  for(int i = 0; i < 100; ++i) {
    b[i] += a[i];
  }
}
```

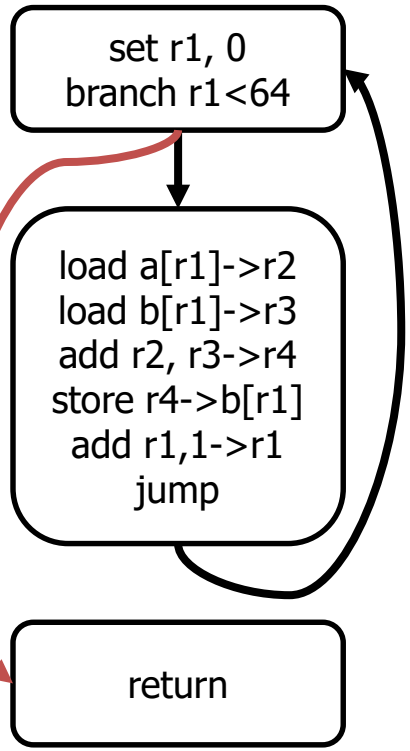
parsing

Abstract Syntax Tree

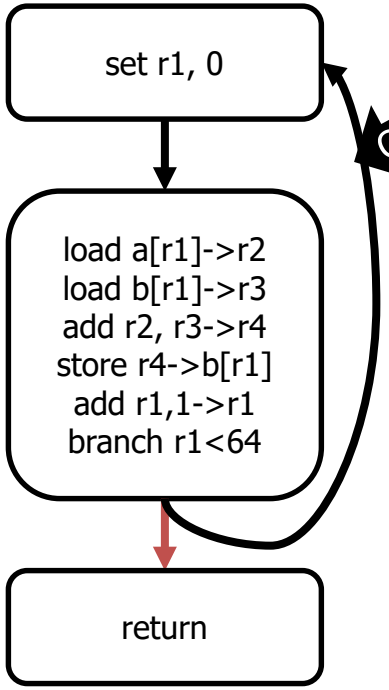


IR Gen

Intermediate Representation: Control Data Flow Graph (CDFG)



Optimization



Code Gen

```
endbr64
lea     0xf(%rsi),%rax
sub     %rdi,%rax
cmp     $0x1e,%rax
mov     $0x0,%eax
jbe     40 <func+0x40>
nopw    %cs:0x0(%rax,%rax,1)

movdqu (%rsi,%rax,1),%xmm0
movdqu (%rdi,%rax,1),%xmm1
paddd  %xmm1,%xmm0
movups %xmm0,(%rsi,%rax,1)
add     $0x10,%rax
cmp     $0x190,%rax
jne     20 <func+0x20>
retq
nop
mov     (%rdi,%rax,1),%edx
add     %edx,(%rsi,%rax,1)
add     $0x4,%rax
cmp     $0x190,%rax
jne     40 <func+0x40>
retq
```

- **Choose registers**
- **Choose specific instructions**
- **Layout code**

Compiler IR

- Like Assembly
- Control flow explicit (no code layout, Infinite registers, Abstract instructions)
- Not specific to the ISA

Question:

1. ISA Objectives?

2. ISA Considerations/Aspects?

What Makes a Good ISA?

1. **Programmability**
   - Easy to express broad range of programs? (human or compiler)
2. **Implementability**
   - Easy to design effective implementations?
   - Where effectiveness is high-performance, low-power, high reliability, low-cost, etc.
3. **Compatibility**
   - Easy to maintain programmability (implementability) as languages and programs evolve?
   - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4,…

1. Programmability

- How easy is it to express programs efficiently?
  - For whom?

- Early: **human**
  - Compilers were terrible, most code was hand-assembled
  - Want: higher-level idioms, coarse grain, similar to language

- Last several decades: **compiler**
  - Optimizing compilers usually generate better code than you or I
  - Want: low-level, simple, fine-grain instructions

Human Programmability

- What makes an ISA easy for a human to program in?
  - Proximity to a high-level language (HLL)
    - Closing the "**semantic gap**"
  - Semantically heavy (CISC-like) insns that capture complete idioms
    - "Access array element", "loop", "procedure call"
    - Example: SPARC **`save`**/**`restore`**
    - Bad example: x86 **`rep movsb`** (copy string)
    - Ridiculous example: VAX **`insque`** (insert-into-queue)
  - "**Semantic clash**":
    - what if you have many high-level languages?
    - what if you have many data-structures to support?

Compiler Programmability

- What makes an ISA easy for a compiler to program in?
  - Low level primitives from which solutions can be synthesized
    - Compilers good at breaking complex structures to simple ones
      - Requires decomposition
    - Not so good at combining simple structures into complex ones
      - Known as "lifting"
      - Requires search, pattern matching
  - Rules of thumb
    - "**principle of least astonishment**"
      - No surprises please – keep it simple
    - Orthogonality / Composability / Regularity
      - All features (e.g. addressing mode) should be independently applicable

2. Implementability

- Any ISA can be implemented
  - But only some ISAs enable efficient implementations

- Classic high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution (more later)

- Certain ISA features make the above difficult
  - Variable instruction lengths/formats: complicate decoding
  - Implicit state: complicates dynamic scheduling
  - Variable latencies: complicates scheduling
  - Difficult to interrupt instructions: complicate many things

  - A solution: High-performance x86 machines dynamically translate complex instructions into simple internal "microops" (translate CISC→RISC, more later)

3. Compatibility

- No-one buys new hardware… if it requires new software
  - IBM established ISA for mainframes; Intel for PCs, ARM for mobile, etc.
  - ISA must remain compatible, no matter what
    - x86 arguably one of the worst ISAs EVER, but survives
    - As does IBM's 360/370/390/z (the *first* "ISA family")
- **Backward compatibility**
  - New processors must support old programs
    - Can't drop features, but can deprecate and emulate
  - Very important
- **Forward (upward) compatibility**
  - Old processors must support new programs (with software help)
    - New processors redefine only previously-illegal opcodes
    - Allow software to detect support for specific new instructions
    - Old processors emulate new instructions in low-level software

The Compatibility Trap

- Easy compatibility requires forethought
  - Temptation: use some ISA extension for 5% performance gain
  - Frequent outcome: gain diminishes, disappears, or turns to loss
    - Must continue to support gadget for eternity

  - Example: register windows (SPARC)
    - Idea: Reduce register spills and fills for function calls by adding: Implicit stack of registers, output registers -> input registers
    - Adds cost and complexity to out-of-order implementations of SPARC

  - Example: branch delay slot (most RISCs)
    - Idea: Eliminates branch hazard in simple 5-stage pipeline by: always executing the instruction after a branch
    - Complicates multi-instruction issue (superscalar)

The Compatibility *Trap Door*

- Compatibility's friends
  - **Trap**: instruction makes low-level "function call" to OS handler
  - **Nop**: "no operation" - instructions with no functional semantics
- Backward compatibility (old programs on new hardware)
  - Handle rarely used but hard to implement "legacy" opcodes
  - Define to trap in new implementation and emulate in software
    - Rid yourself of some ISA mistakes of the past
    - Problem: performance suffers for legacy codes
- Forward compatibility (new programs on old hardware)
  - Reserve sets of trap & nop opcodes (don't define uses)
  - Add ISA functionality by overloading traps
    - Firmware or OS patch
  - Add ISA hints by overloading nops

Blocking the Compatibility Trap Door

- Temptation:
  - Define "unused" instruction fields as "don't cares"
    - E.g., MIPS "shift length" field in an "add" instruction
  - Simplifies hardware logic needed to decode instructions

- Problem:
  - Can't use "unused" values for new instructions
  - Same problem for special registers (e.g., Interrupt status register)

- Solution:
  - Define all bits (usually to be zero).
  - Undefined instructions are trapped

Aspects of ISAs

ISA Aspects

- **Execution Model (Implicit vs Explicit Dependences)**
  - **VonNeuman vs Dataflow**
- **Implicit vs Explicit Parallelism Specification**
  - **Scalar vs Vector vs VLIW**
- Format (Variable vs Fixed-width)
- Operand Model (where is temporary data stored)
  - Register, Stack, Accumulator
- Addressing Modes (how do you specify addresses)
  - Register-Indirect, Displacement, Index-base
- Memory Abstractions (physical/virtual addr. space)
- Control (condition, branch/predication)
- Details: Register file size, Datatypes (float vs int, bitwidth), Instruction Complexity -- loops

Execution Model

- How to decide when to execute an instruction?
- Option 1: Execute them in order (some predefined order)
  - Most high level imperative languages are anyways sequential…
  - Use original program order
- Option 2: Let dependences decide
  - Instructions trigger the execution of dependent instructions
  - Parallel execution "automatic" as instructions execute
- Option 3: Predetermine everything
  - Decide in advance the exact timing (and resource use) of every instruction

Von Neumann (sequential) Model

- Implicit model of modern commercial ISAs (ARM,x86, etc)
  - Called von Neuman, but in ENIAC design before*
- Basic feature: the **program counter (PC)**
  - Defines **total order** on dynamic instruction
    - Next PC is PC++ unless insn says otherwise
    - Branch instructions change PC
  - Order and **named storage** define computation
    - Value flows from insn X to Y via storage A iff…
    - Y names A as input…
    - X is the youngest previous insn to name A as output (in total order)

* Probably should be called Eckert/Mauchly Model…

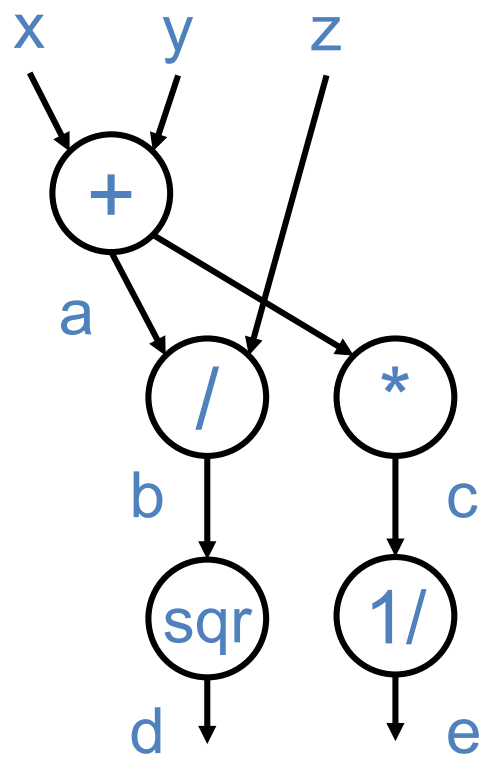
Dataflow Model

- Dependences encoded explicitly in the ISA
  - Eg. X->Y is explicit (as opposed to implicit communication through general purpose registers/memory)
- Dataflow firing rule:
  - Instructions execute when all operands are ready
  - Upon completion, data values are forwarded to instructions which consume them.
- Dataflow direction:
  - Backwards: Instruction specifies producers/source operands, and should retrieve data from them.
  - Forwards: Instruction specifies consumers, and delivers data directly to them.
    (forward is arguably better because distribution is synchronized)
- Lacks PC: Unnecessary because dependencies sufficient

Von Neuman vs Dataflow

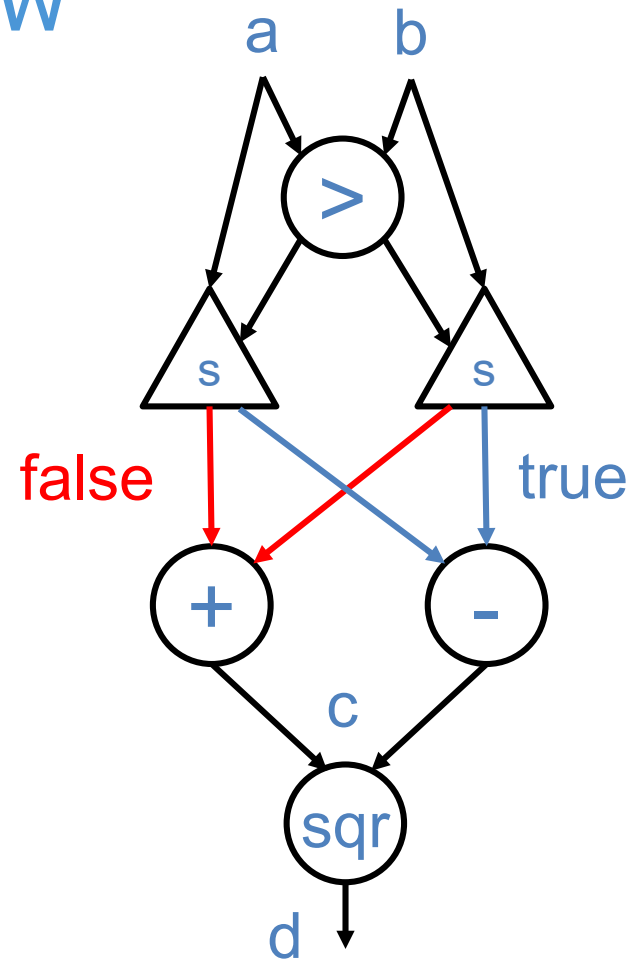
PC →

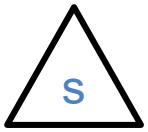
```
a = x + y;
b = z / a;
c = b * a;
d = sqrt(b);
e = 1 / c;
```



Control Flow in Dataflow

```
if(a > b) {
  c = a - b;
} else {
  c = a + b;
}
d = sqrt(c);
```

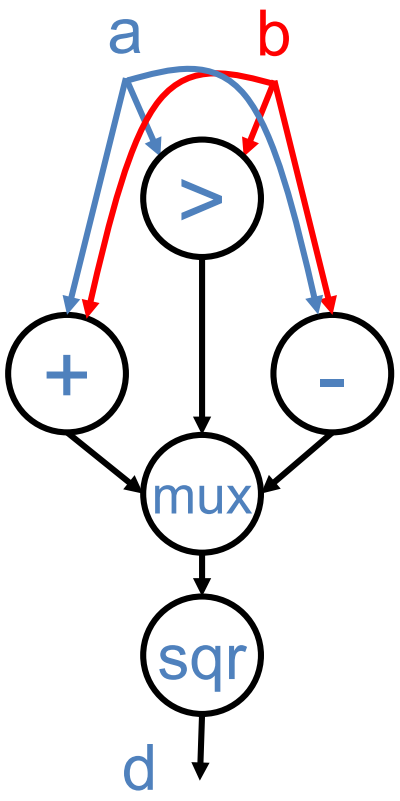




Switch(data input**,** control input **->** output1**,** output2**) {**
  if**(**control input**)** forward data to output1
  else forward to output 2
}

Control Flow in Dataflow (alternate)

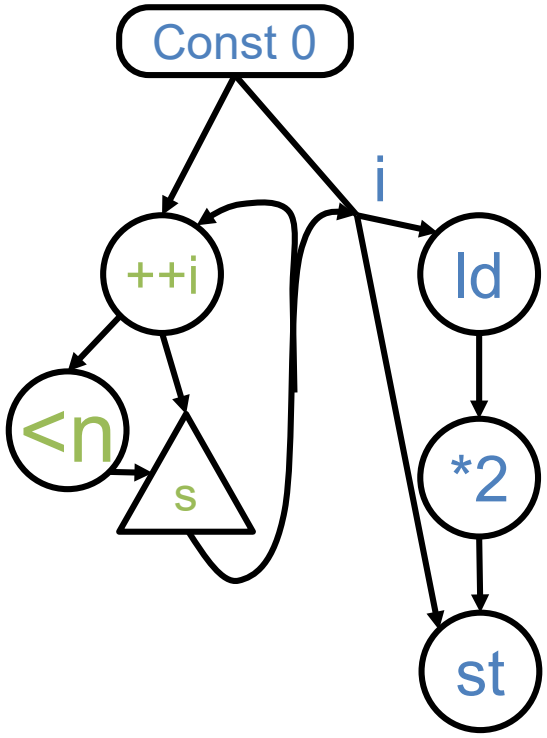
```
if(a > b) {
  c = a - b;
} else {
  c = a + b;
}
d = sqrt(c);
```



Q: What are the benefits of switch versus mux?

Control Flow in Dataflow (loops)

```
for(int i = 0; i < n; ++i) {
  b[i] = a[i] * 2;
}
```



Q: What's the critical path of this loop?
Q: What if left portion (control loop) executes much faster than the right hand side?

Pipelined Dataflow Execution

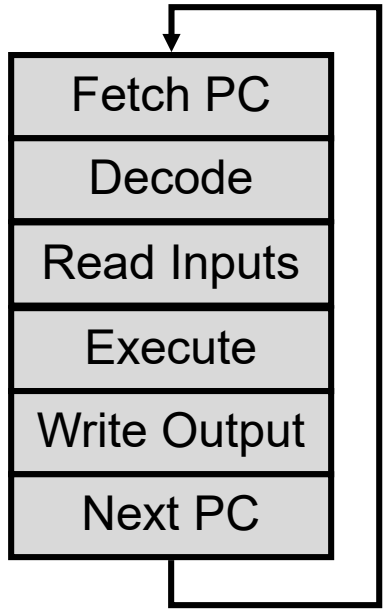
- Dataflow needs mechanism to distinguish multiple iterations of the same instructions.
  - Option 1: Ordering
    - Software enforces that operands will arrive in the correct order regardless of the latencies of instructions
    - E.g. add an instruction at every merge point to choose which location will be the producer
      - Merge point (where an operand can be defined by >1 producing instruction)
  - Option 2: Tagging
    - Values are **tagged** or annotated with a number indicating which instance they are.
      - Eg. Loop iteration can be a tag
    - Only operands with the same tag can be **matched**
    - Requires some more complex hardware

Many more fun issues in Dataflow

- Control flow
- Pipelining
- Enabling sequential memory semantics
- Control speculation
- Memory speculation
- Speculative scheduling (back-to-back execution of dependent instructions)
- Resource Scheduling (static / dynamic)
- Parallelization / vectorization

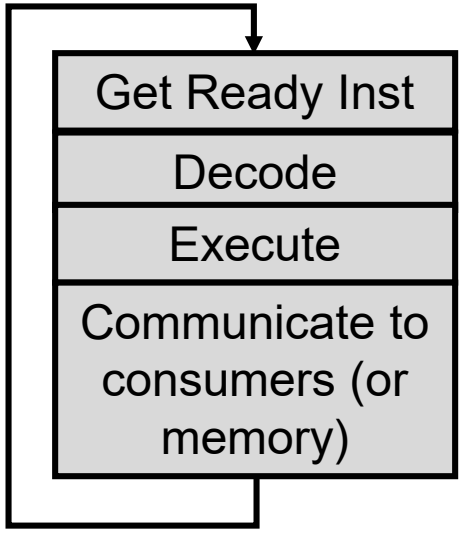
Implementation Implications

- Von Neuman Implementation
  - Instruction X finishes before insn X+1 starts



Canonical Format

- Dataflow Implementation
  - Just choose one/some of the ready instructions



Canonical Format

VonNeumann/Dataflow Tradeoffs

- Von Neuman
  - Same sequential model as imperative programs
    -- easy to reason about, easy to translate (not parallel)
  - Very easy to "pause" execution at any point in the program
  - Total ordering of instructions makes speculation recovery simpler
  - Total ordering of memory eases dependence/alias detection
- Dataflow
  - Multiple instructions may be ready simultaneously -> parallelism
  - Much harder to pause
    - Many instructions active – more state to save on context switch
    - Hard to reason about (debugging)
  - Related problem: unbounded state…
- Solution is Hybrid!
  - Von Neumann Outside / Dataflow Inside
    - Bounded state, can still recover, still get parallelism
    - Eg. Out-of-order cores; dataflow processors: Trips/Wavescalar

- Expressing *dependences* in the ISA (e.g. via dataflow) is one way to expose parallelism to microarchtiecture

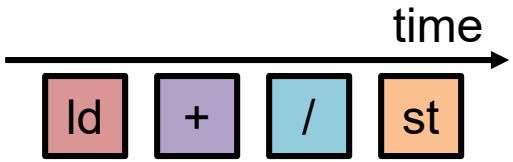
- Alternate way to expose parallelism:
  - **Express independence ?**

Implicit vs Explicit Parallelism

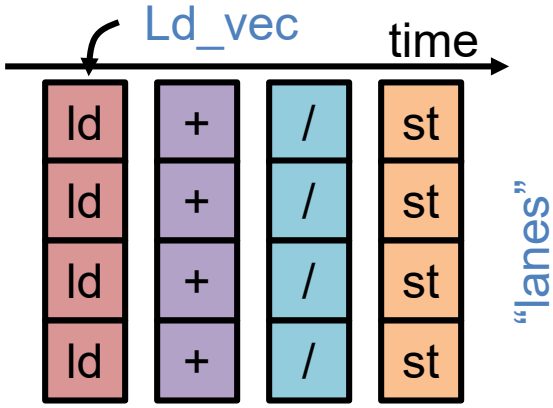
Implicit vs Explicit Parallelism

```
for i= 0 -> N
  b[i] = (a[i]+1)/2
```

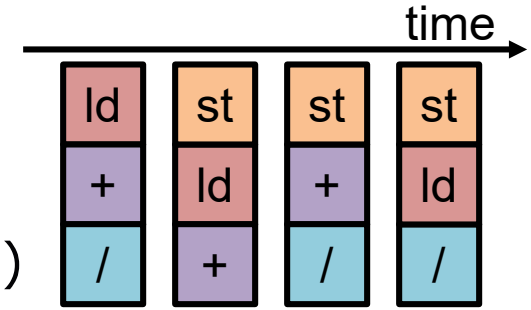
- Possiblitiy 1: Scalar Instructions
  - No parallelism explicitly
  - Register Datatype: Scalar Types



- Possibility 2: Vector Instrutcions
  - Registers: Wide Vectors (eg. 2-128 elem)
  - Single instruction over multiple data (SIMD*)
  - Advantages:
    - Less instructions
    - Parallelism explicit (h/w simpler)
  - Issue: Need data-level parallelism…



- Possibility 3: VLIW: Very long inst. word
  - Instruction packets specify parallel insns
  - Registers: Scalar (more-or-less)
  - Advantage: Parallelism explicit (h/w simpler)
  - Issue: Need (static) ILP and regular memory :)



- Worst part of 2&3: Many Compiler Challenges!

*This is the normal time to introduce "Flynn's" taxonomy, but it is pretty much useless so lets not go there…

State-of-the-Art in Implicit vs Explicit

- Possible to combine above techniques in a single ISA, or only use them separately.
- Today there are basically two main approaches for CPUs:
  - Scalar ISA + vector extensions
    - Extensions useful for when lots of DLP
    - Eg. X86 SSE/MMX/AVX, ARM Neon, IBM AltiVec
  - "Pure" VLIW Processors
    - Useful for specialized tasks like signal processing
    - Eg. TI DSPs, also Intel IA64
- Fancier SIMD: SIMT – Single Instruction Multiple Thread
  - Basic Idea: broadcast one instruction at a time to multiple threads, but each thread has its own logical context (so threads can diverge)
  - This is used by GPUs, we will discuss in detail in final lectures…

ISA Aspects

- Execution Model (Implicit vs Explicit Dependences)
  - VonNeuman vs Dataflow
- Implicit vs Explicit Parallelism Specification
  - Scalar vs Vector vs VLIW
- **Format (Variable vs Fixed-width)**
- **Operand Model (where is temporary data stored)**
  - **Register, Stack, Accumulator**
- **Addressing Modes (how do you specify addresses)**
  - **Register-Indirect, Displacement, Index-base**
- **Memory Abstractions (physical/virtual addr. space)**
- Control (condition, branch/predication)
- Details: Register file size, Datatypes (float vs int, bitwidth), Instruction Complexity -- loops

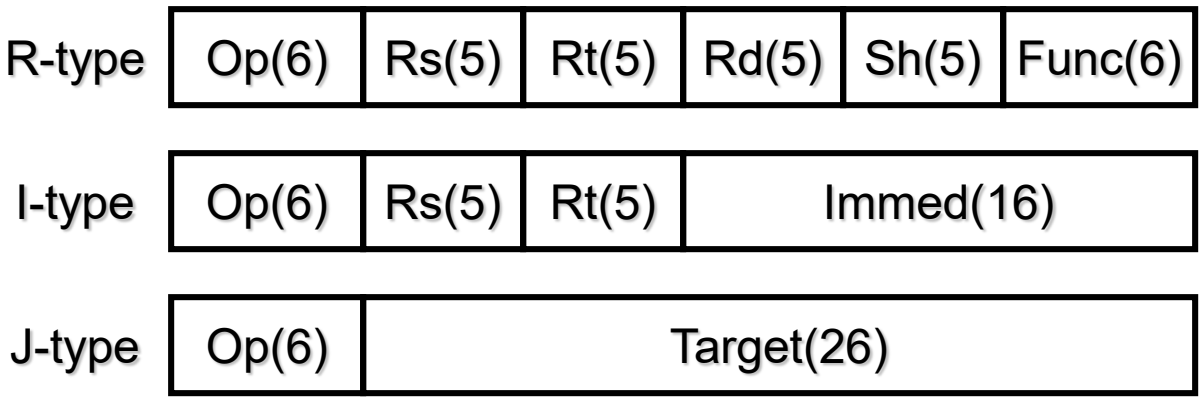
Format

- **Length**
  - Fixed length
    - Most common is 32 bits
    - \+ Simple implementation: compute next PC using only PC
    - – Code density: 32 bits to increment a register by 1?
      - – x86 can do this in one 8-bit instruction
  - Variable length
    - – Complex implementation
    - \+ Code density
  - Compromise: two lengths
    - MIPS16 or ARM's Thumb
- **Encoding**
  - A few simple encodings simplify decoder implementation

Simple Example: MIPS Format

- Length
  - 32-bits

- Encoding
  - 3 formats, simple encoding
  - Q: how many instructions can be encoded? A: 64? 127? 4096?



Not Simple Example: X86 Format

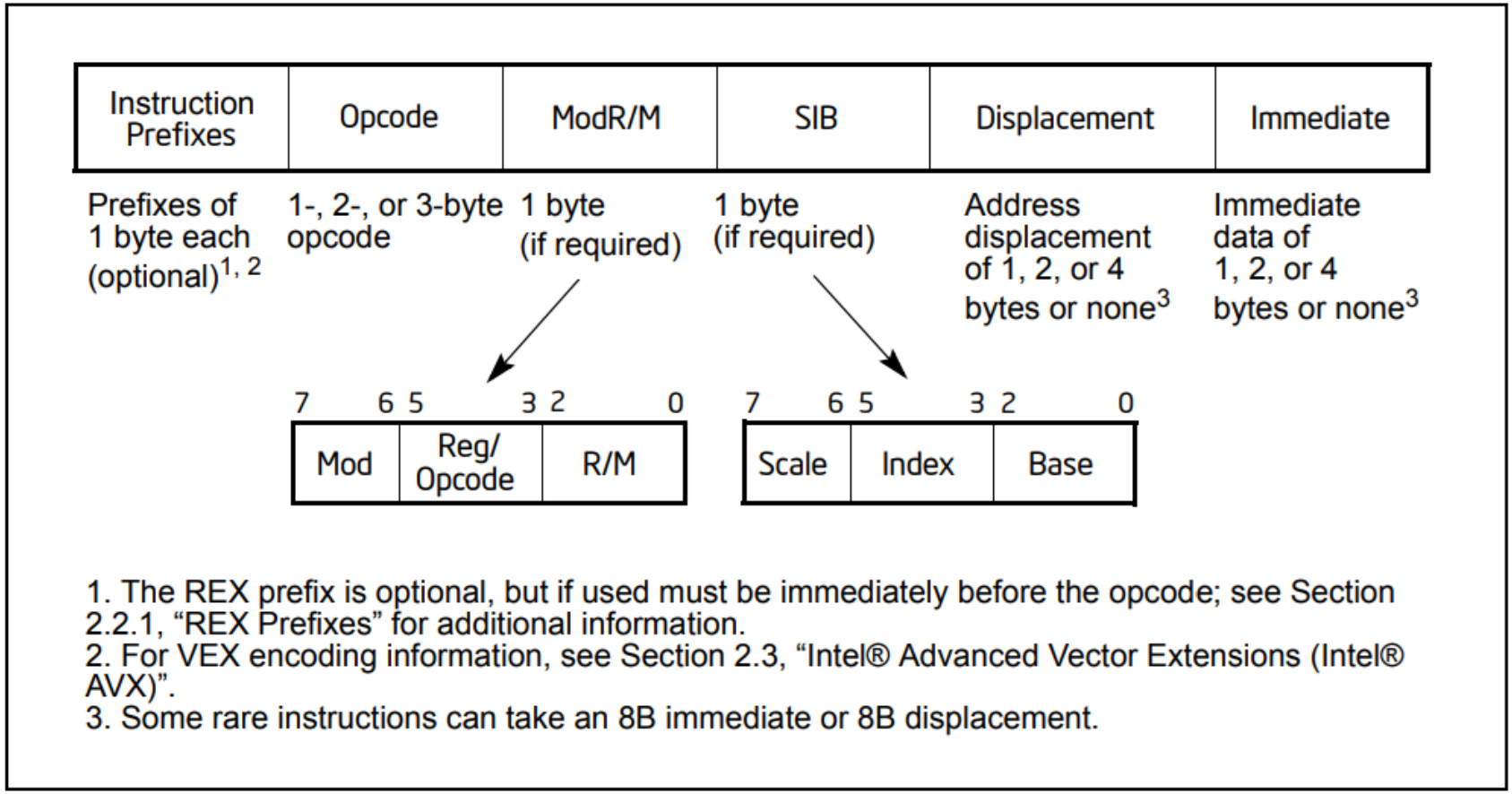


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

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

Operand Model:

What to do with temporaries?

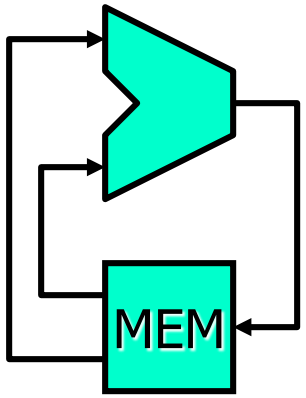
Operand Model: Memory Only

- Where (other than memory) can operands come from?
  - And how are they specified?
  - Example: A = B + C
  - Several options

- **Memory only**

  `add B,C,A` mem[A] = mem[B] + mem[C]

  - Not practical

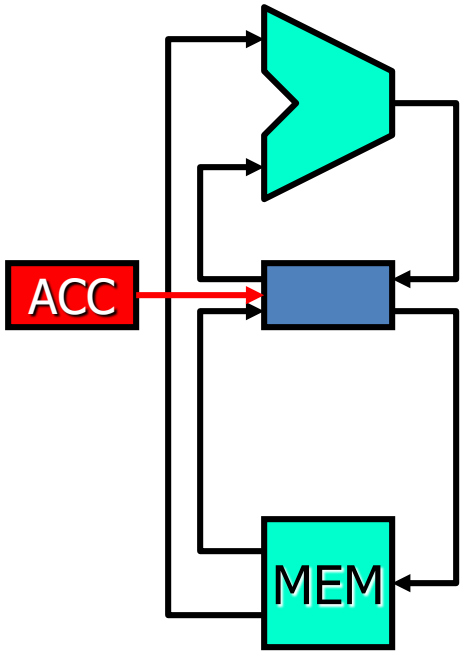


(A,B,C are pointers)

Operand Model: Accumulator

- **Accumulator**: implicit single element storage

```
load B          ACC = mem[B]
add C           ACC = ACC + mem[C]
store A         mem[A] = ACC
```

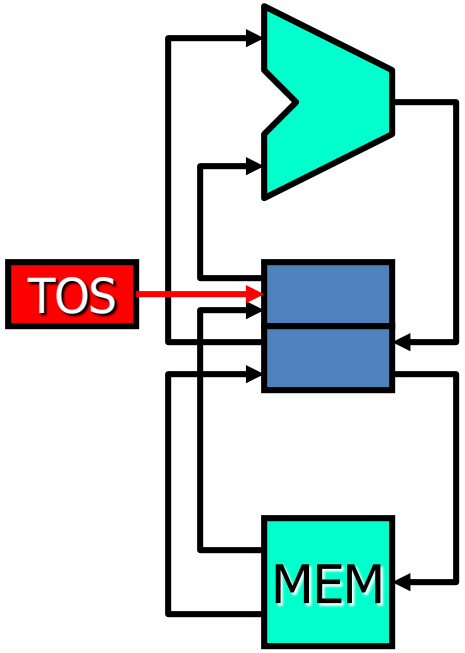


(A,B,C are pointers)

Operand Model: Stack

- **Stack**: TOS implicit in instructions

`push B`	stk[TOS++] = mem[B]
`push C`	stk[TOS++] = mem[C]
`add`	stk[TOS++] = stk[--TOS] + stk[--TOS]
`pop A`	mem[A] = stk[--TOS]



(A,B,C are pointers)

Operand Model: Registers

- **General-purpose register**: multiple explicit accumulator

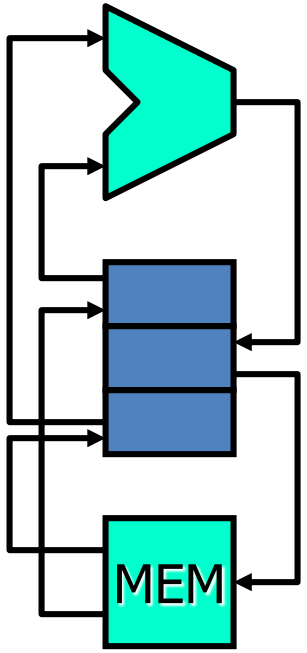
```
load B,R1        R1 = mem[B]
add C,R1         R1 = R1 + mem[C]
store R1,A       mem[A] = R1
```

(eg. x86)

- **Load-store**: GPR and only loads/stores access memory

```
load B,R1        R1 = mem[B]
load C,R2        R2 = mem[C]
add R1,R2,R1     R1 = R1 + R2
store R1,A       mem[A] = R1
```

(eg. MIPS)



(A,B,C are pointers)

Operand Model Pros and Cons

- Metric I: **static code size**
  - Number of instructions needed to represent program, size of each
  - Want many implicit operands, high level instructions
  - Good $\rightarrow$ bad: accumulator, stack, GP-register, load-store
- Metric II: **data memory traffic**
  - Number of bytes move to and from memory (including cache)
  - Want as many long-lived operands in on-chip storage
  - Good $\rightarrow$ bad: load-store / GP-register, stack, accumulator
- Metric III: **cycles per instruction**
  - Want short (1 cycle?), little variability, few nearby dependences
  - Good $\rightarrow$ bad: load-store, GP-register, stack, accumulator
- Upshot: most new ISAs are load-store (or GP-register)

How Many Registers?

- Why not have as many as possible registers?
  - One reason registers are faster is that there are **fewer of them**
    - Smaller is faster in h/w (shorter wires, simpler decoder, etc)
  - Another is that they are **directly addressed** (no address calc)
    - More of them, means larger specifiers
    - Fewer registers per instruction or indirect addressing
  - **Not everything can be put in registers**
    - Structures, arrays, anything pointed-to
    - Although compilers are getting better at putting more things in
  - More registers means **more saving/restoring**
    - Context switching gets more expensive

- Modern ISAS:
  - X86:8, X86_64: 16 GP + 16 FP
  - MIPS: 32
  - IA64/Itanium: 128
  - RISCV: 32 (16 embedded variant)

Virtual Address Size

- What is an n-bit processor?
  - Support memory size of 2^n
  - Alternative (wrong) definition: size of operators
- Determines maximum size of addressable (usable) memory
  - Most high-perf devices are 64 bit (server/cell/laptop/desktop)
  - Most implementations limited to 40-50 bits
    - X86-64 uses 48-bit virtual address space
- A pain to overcome too-small virtual address space
  - x86 evolution:
    - 12-bit (4004), 14-bit (8008), 16-bit (8086), 24-bit (80286),
    - 32-bit + protected memory (80386)
    - 64-bit (AMD's Opteron & Intel's EM64T Pentium4)
- Size of Addressability: smallest addressable unit
  - Most architectures use 8-bits (byte-level)

Memory Addressing

- **Addressing mode:** way of specifying address
  - Used in memory-memory or load/store instructions in register ISA
- Examples
  - **Register-Indirect:** R1=mem[R2]
  - **Displacement:** R1=mem[R2+immed]
  - **Index-base:** R1=mem[R2+R3]
  - **Memory-indirect:** R1=mem[mem[R2]]
  - **Auto-increment:** R1=mem[R2], R2= R2+1
  - **Auto-indexing:** R1=mem[R2+immed], R2=R2+immed
  - **Scaled:** R1=mem[R2+R3*immed1+immed2]
  - **PC-relative:** R1=mem[PC+imm]
- What high-level program idioms are these used for?
- What implementation impact? What impact on insn count?

Two More Addressing Issues

- **Access alignment**: address % datatype_size == 0?
  - Aligned: `load-32bit @XXXX00, load-16bit @XXXXX0`
  - Unaligned: `load-32bit @XXXX10, load-16bit @XXXXX1`
  - Unaligned uncommon because program languages typically align data structures by default.

- Question: what to do with unaligned accesses?
  - Disallow (unaligned operations are illegal)
    - MIPS takes this route
    - Use regular instructions: "Load, shift, load, shift, and"
  - Trap to software routine?
  - Support in hardware?
    - x86 allows regular loads/stores to be unaligned
    - Unaligned access still slower, adds significant hardware complexity (Straddling cache lines)

ISA Aspects

- Execution Model (Implicit vs Explicit Dependences)
  - VonNeuman vs Dataflow
- Implicit vs Explicit Parallelism Specification
  - Scalar vs Vector vs VLIW
- Format (Variable vs Fixed-width)
- Operand Model (where is temporary data stored)
  - Register, Stack, Accumulator
- Addressing Modes (how do you specify addresses)
  - Register-Indirect, Displacement, Index-base
- Memory Abstractions (physical/virtual addr. space)
- **Control (condition, branch/predication)**
- **Details: Register file size, Datatypes (float vs int, bitwidth), Instruction Complexity -- loops**

Control Instructions

- One issue: **testing for conditions**
  - Option I: **compare and branch insns (e.g. RISCV)**

```
branch-less-than R1,10,target
```

    + Simple, – two ALUs: one for condition, one for target address

  - Option II: **implicit condition codes (e.g. x86)**
    - `subtract R2,R1,10  // sets "negative" CC`

```
branch-neg target
```

    + Condition codes set "for free", – implicit dependence is tricky

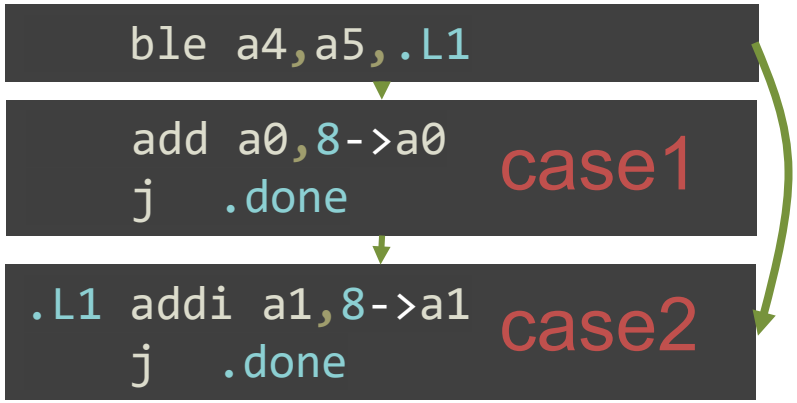
  - Option III: **condition registers, separate branch insns (ARM)**

```
compare R1,10 -> R2
branch-not-equal-zero R2,target
```

    – Additional instructions, + one ALU per, + explicit dependence

Control Instructions II

- Unpredictable branches are bad – can we eliminate?
- Sure: Predication
  - Turns control dependence into data dependence
  - Instructions have a predicate operand that determines whether they are executed – so no need to jump to different code
  - Predicate registers can be set by compare instructions
- Example: if(a4 < a5) {a0++;} else {a1++;}




```
set_le   a4,a5->p1
p1==false: addi a0,a0,8
p1==true: addi a1,a1,8
j       .done
```

- What's the tradeoff, when would you want to use this?
- Common feature (x86 has limited support, ARM general support)

Control Instructions III

- Another issue: support for procedure calls?
  - Call: Push ret. Address to stack (memory), and jump
  - Link (remember) address of calling insn + 4 so we can return to it

- MIPS
  - jump-and-link: **`jalr Rs, Rd`**
    - Rd = PC + 8
    - PC = Rs
- X86
  - Call
    - target can be displacement, absolute, absolute indirect…
    - Put return address on stack (memory pointed to by stack pointer)
  - Ret
    - Return to location on head of stack

Control Instructions IV

- Another issue: **computing targets**
  - Option I: **PC-relative**
    - Position-independent within procedure
    - Used for branches and jumps within a procedure
  - Option II: **Absolute**
    - Position independent outside procedure
    - Used for procedure calls
  - Option III: **Indirect** (target found in register/memory)
    - Needed for jumping to dynamic targets
    - Used for returns, dynamic procedure calls, switches

  - How far do you need to jump?
    - Typically not so far within a procedure (they don't get that big)
    - Calls/Returns can be longer

ISA Aspects

- Execution Model (Implicit vs Explicit Dependences)
  - VonNeuman vs Dataflow
- Implicit vs Explicit Parallelism Specification
  - Scalar vs Vector vs VLIW
- Format (Variable vs Fixed-width)
- Operand Model (where is temporary data stored)
  - Register, Stack, Accumulator
- Addressing Modes (how do you specify addresses)
  - Register-Indirect, Displacement, Index-base
- Memory Abstractions (physical/virtual addr. space)
- Control (condition, branch/predication)
- Details: Datatypes (float vs int, bitwidth), Endianness, Encoding

Economic Concerns: License-ability?

- X86:
  - Very limited and expensive
    - pre-586 subset is not licensable – cause its *old*
  - Licensed to handful of companies: AMD, VIA
  - AMD X86-64 extension licensed to Intel :)
- ARM:
  - Easy to obtain license
  - License fee: $1M - $10M
  - Royalty: 0.5 - 2% of chip selling price
- RISCV:
  - Fully open source – no fees

IP	Royalty
ARM7/9/11	1.0% - 1.5%
ARM Cortex A-series	1.5% - 2.0%
ARMv8 Based Cortex A-series	2.0% +
Mali GPU	0.75% - 1.25%
Physical IP Package (POP)	0.5%

Source: anandtech.com (2013)

The RISC vs. CISC Debate

RISC and CISC

- **RISC**: reduced-instruction set computer
  - Coined by Patterson in early 80's
  - RISC-I (Patterson), MIPS (Hennessy), IBM 801 (Cocke)
  - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC
- **CISC**: complex-instruction set computer
  - Term didn't exist before "RISC"
  - Examples: x86, VAX, Motorola 68000, etc.

- Philosophical war (one of several) started in mid 1980's
  - RISC "won" the technology battles
  - CISC won the high-end commercial war (1990s to today)
    - Compatibility a stronger force than anyone (but Intel) thought
  - RISC won the embedded computing war, and (if you count ARM as RISC) it won the mobile computing war

The Context

- Pre 1980
  - Bad compilers (so assembly written by hand)
  - Complex, high-level ISAs (easier to write assembly)
  - Slow multi-chip implementations
- Late 1970s/Early 1980s
  - Moore's Law makes single-chip microprocessor possible…
    - **…but only for small, simple ISAs**
  - Performance advantage of this "integration" was compelling
  - Compilers had to get involved in a big way
- **RISC manifesto**: create ISAs that…
  - **Simplify single-chip implementation**
  - **Facilitate optimizing compilation**

The RISC Design Tenets

- **"Single-cycle" execution** (pipelining for inorder core)
  - CISC: many multicycle operations
- **Hardwired control**
  - CISC: microcoded multi-cycle operations
- **Load/store architecture**
  - CISC: register-memory and memory-memory
- **Few memory addressing modes**
  - CISC: many modes
- **Fixed-length instruction format**
  - CISC: many formats and lengths
- **Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance
- **Many registers** (compilers are better at using them)
  - CISC: few registers

CISCs and RISCs

- The CISCs: x86, VAX (**V**irtual **A**ddress e**X**tension to PDP-11)
  - Variable length instructions: 1-321 bytes!!!
  - 14 registers + PC + stack-pointer + condition codes
  - Data sizes: 8, 16, 32, 64, 128 bit, decimal, string
  - Memory-memory instructions for all data sizes
  - Special insns: **`crc`**, **`insque`**, **`polyf`**, and a cast of hundreds
  - x86: "Difficult to explain and impossible to love"
- The RISCs: MIPS, PA-RISC, SPARC, PowerPC, Alpha, ARM
  - 32-bit instructions
  - 32 integer registers, 32 floating point registers, load-store
  - 64-bit virtual address space
  - Few addressing modes
  - Why so many basically similar ISAs? Everyone wanted their own

The Debate

- RISC argument
  - CISC is fundamentally handicapped
  - For a given technology, RISC implementation will be better (faster)
    - Current technology enables single-chip RISC
    - When it enables single-chip CISC, RISC will be pipelined
    - When it enables pipelined CISC, RISC will have caches
    - When it enables CISC with caches, RISC will have next thing...

- CISC rebuttal
  - CISC flaws not fundamental, can be fixed with more transistors
  - Moore's Law will narrow the RISC/CISC gap (true)
    - Good pipeline: RISC = 100K transistors, CISC = 300K
    - By 1995: 2M+ transistors had evened playing field
  - Software costs dominate, **compatibility** is paramount

RISC vs CISC Performance Argument

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- **CISC** (Complex Instruction Set Computing)
  - Reduce "instructions/program" with "complex" instructions
    - But tends to increase CPI or clock period
  - Easy for assembly-level programmers, good code density
- **RISC** (Reduced Instruction Set Computing)
  - Improve "cycles/instruction" with many single-cycle instructions
  - Increases "instruction/program", but hopefully not as much
    - **Help from smart compiler**
  - Perhaps improve clock cycle time (seconds/cycle)
    - **via aggressive implementation allowed by simpler insn**

Current Winner (Server/Desktop): x86

- x86 was first 16-bit chip by ~2 years
  - IBM put it into its PCs because there was no competing choice
  - Rest is historical inertia and "financial feedback"
- Moore's law has helped Intel in a big way
  - Many engineering problems can be solved with more transistors
- Complex architecture due to evolution over time
  - Typical of many older ISAs, e.g. IBM 360/370/390
  - Started as 16-bit microprocessor (later, 32-bits)
  - Upward compatible from 8080 (accumulator-based)
- Apple challenging this victory with ARM-based M1,M2, etc.

Current Winner (mobile/embed.): ARM

- ARM (Advanced RISC Machine)
  - First ARM chip in mid-1980s (from Acorn Computer Ltd).
  - Over 230 billion units sold
  - Low-power and embedded devices, and now phones
- 32-bit RISC ISA
  - 16 registers
  - Many addressing modes (for example, auto increment)
  - Condition codes, each instruction can be predicated
- Multiple compatible implementations
  - Intel's X-scale (was DEC's)
  - Others: Freescale (was Motorola), IBM, Texas Instruments, Nintendo, STMicroelectronics, Samsung, Sharp, Philips, etc.
- "Thumb" 16-bit wide instructions
  - Increase code density

Intel's Compatibility Trick: RISC Inside

- 1993: Intel wanted out-of-order execution in Pentium Pro
  - Hard to do with a coarse grain ISA like x86
- Solution? Translate x86 to RISC **μops** in hardware

  ```
  push $eax
  ```

  becomes (we think, uops are proprietary)

  ```
  store $eax, -4($esp)
  addi $esp,$esp,-4
  ```

  + Processor maintains **x86 ISA externally for compatibility**
  + But executes **RISC μISA internally for implementability**
  - Given translator, x86 almost as easy to implement as RISC
    - Intel implemented out-of-order before any RISC company
    - Also, OoO also benefits x86 more (because ISA limits compiler)
  - Idea co-opted by other x86 companies: AMD and Transmeta
  - Different **μops** for different designs
    - **Not part of the ISA specification**, not publicly disclosed

Potential Micro-op Scheme (1 of 2)

- Most instructions are a **single** micro-op
  - Add, xor, compare, branch, etc.
  - Loads example: mov -4(%rax), %ebx
  - Stores example: mov %ebx, -4(%rax)
- Each memory operation adds a micro-op
  - "addl -4(%rax), %ebx" is two micro-ops (load, add)
  - "addl %ebx, -4(%rax)" is three micro-ops (load, add, store)
- What about address generation?
  - Simple address generation is generally part of single micro-op
    - Sometime store addresses are calculated separately
  - More complicated (scaled addressing) might be separate micro-op

Potential Micro-op Scheme (2 of 2)

- Function call (CALL) – 4 uops
  - Get program counter, store program counter to stack, adjust stack pointer, unconditional jump to function start
- Return from function (RET) – 3 uops
  - Adjust stack pointer, load return address from stack, jump to return address
- Other operations
  - String manipulations instructions
    - For example STOS is around six micro-ops, etc.

- Again, this is just a basic idea, the exact micro-ops are specific to each chip

Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures

- Goal & Position?
  - Determine does RISC vs CISC matter? (no, just microarchitecture)
- Why now, what are the challenges?
  - Factors independent of ISA and microarchitecture: technology, memories, OS, compiler backend, workloads.
- Methodology?
  - Native execution for performance/uarch events, Physically measure power (Wattsup), deal with technology scaling analytically, emulation/simulation for instruction mix
- Interesting Findings?

Crux of argument

- Performance differs a lot….

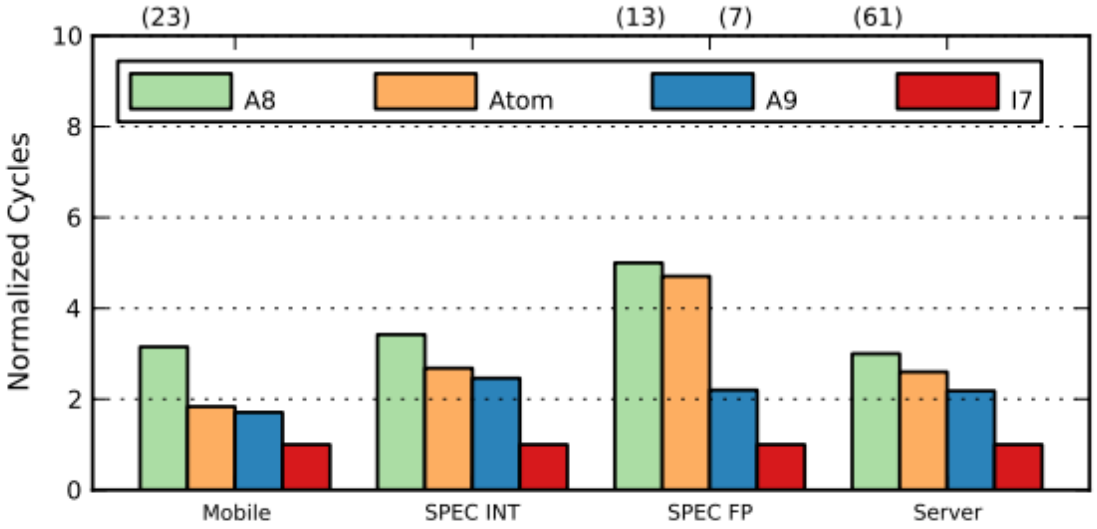


Figure 3. Cycle Count Normalized to i7.

- And so does the power….

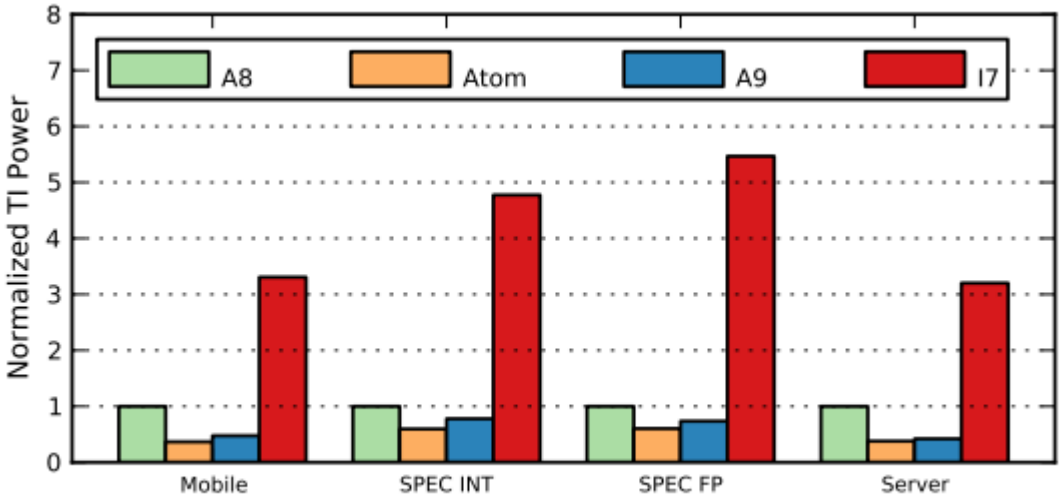
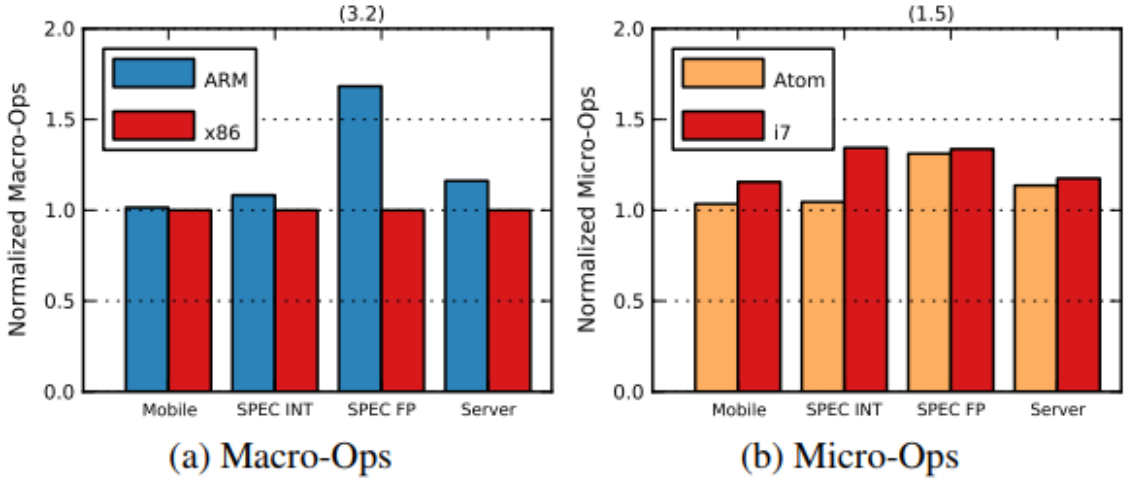


Figure 12. Tech. Independent Avg. Power Normalized to A8.

ISA doesn't seem to be playing a role…

- # Macro-ops and micro-ops are similar



(a) Macro-Ops (b) Micro-Ops

Figure 4. Instructions Normalized to i7 macro-ops.

- Instruction mix is pretty similar

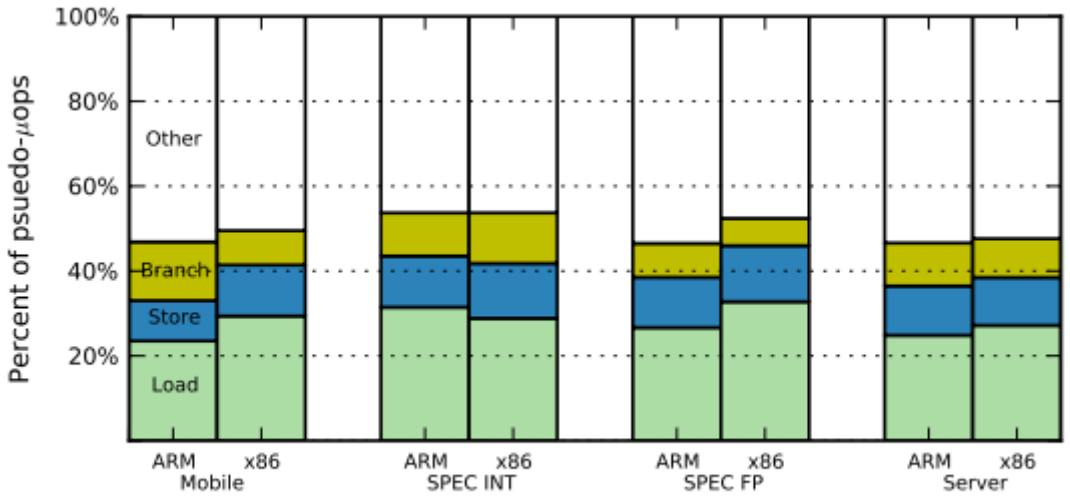


Figure 5. Instruction Mix (Performance Counters).

But microarchitecture plays a large role:

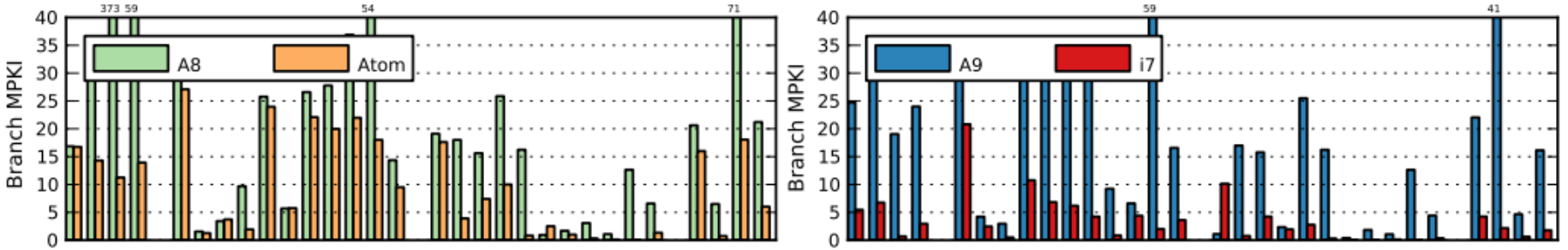


Figure 8. Branch Misses per 1000 ARM Instructions.

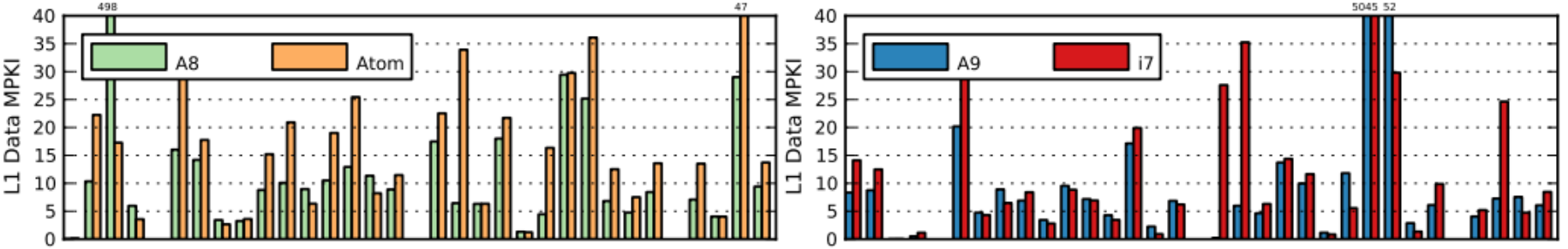
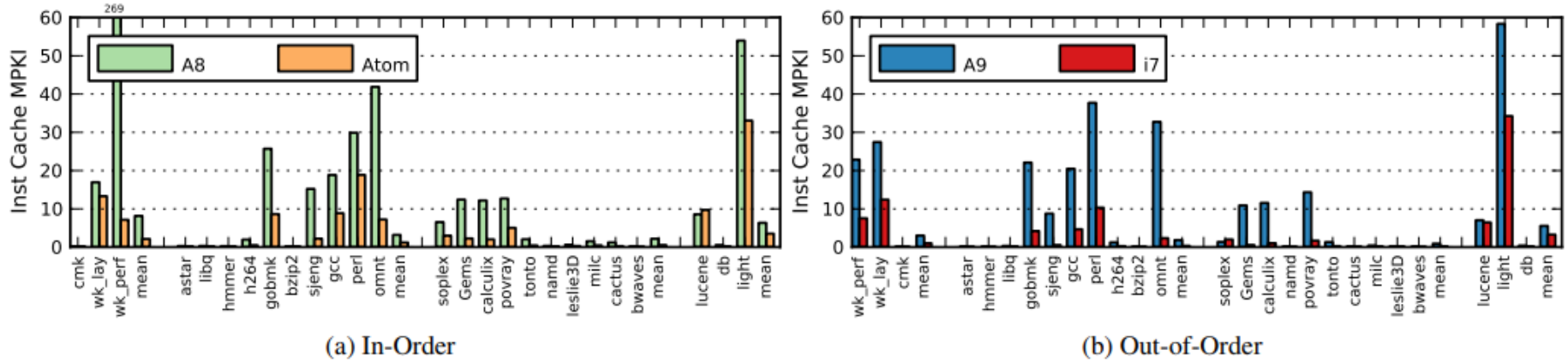


Figure 9. Data L1 Misses per 1000 ARM Instructions.



(a) In-Order

(b) Out-of-Order

At least one difference…

- Code density

Table 6. Instruction Size Summary.

		(a) Binary Size (MB)		(b) Instruction Length (B)	
		ARM	x86	ARM	x86
Mobile	Minimum	0.02	0.02	4.0	2.4
	Average	0.95	0.87	4.0	3.3
	Maximum	1.30	1.42	4.0	3.7
Desktop INT	Minimum	0.53	0.65	4.0	2.7
	Average	1.47	1.46	4.0	3.1
	Maximum	3.88	4.05	4.0	3.5
Desktop FP	Minimum	0.66	0.74	4.0	2.6
	Average	1.70	1.73	4.0	3.4
	Maximum	4.75	5.24	4.0	6.4
Server	Minimum	0.12	0.18	4.0	2.5
	Average	0.39	0.59	4.0	3.2
	Maximum	0.47	1.00	4.0	3.7

Power Struggles

- Any opinions on validity of results?
  - My opinion: Nobody was disputing in the first place that microarchitecture makes a big difference…
  - Microarchitectures are similar only in terms of high-level parameters…

- Could you do the same study on a simulator?
  - Advanatage: Microarchtiectures can be made to be identical (just change gem5 ISA between ARM & RISC)
  - Disadvantage: Simulator validity/bugs become a big problem. :)

Will ISAs (and ISA research) continue to be important?

RISCV vs CISC

- Does "quality" of ISA actually matter?
  - Not for performance (mostly)
    - Mostly comes as a design complexity issue
    - Insn/program: everything is compiled, compilers are good
    - Cycles/insn and seconds/cycle: μISA, many other tricks
  - What about power efficiency? *Maybe*
    - ARMs are most power efficient today…
      - …but Intel is moving x86 that way (e.g, Intel's Atom)
    - Open question: can x86 be as power efficient as ARM?
- Does "nastiness" of ISA matter?
  - Well, only compiler writers and hardware designers see it..
- Even compatibility is not what it used to be
  - Software emulation, dynamic/load-time compilation
  - Open question: will "ARM compatibility" be the next x86?
- RISC vs CISC – "missing forest for the trees"

Summary

- What is an ISA?
  - A functional contract
- What makes a good ISA
  - {Programm|Implement|Compat}-ability
  - Enables high-performance
  - At least doesn't get in the way
- ISAs similar in many high-level ways
  - But many design choices in details
  - Two "philosophies": CISC/RISC
    - Difference is blurry – maybe there's better CISC in the future
  - Other distinctions more important, dataflow, vector, etc…
- Compatibility is a powerful force
  - Tricks: binary translation, μISAs mitigation

Hidden Bonus Slides

Ultimate Compatibility Trick

- Support old ISA by…
  - …having a simple processor for that ISA somewhere in the system
  - How first Itanium supported x86 code
    - x86 processor (comparable to Pentium) on chip
  - How PlayStation2 supported PlayStation games
    - Used PlayStation processor for I/O chip **& emulation**

More X86 RISC Tricks

- x86 code is becoming more "RISC-like"
  - In 32-bit to 64-bit transition, x86 made two key changes:
    - Double number of registers, better function calling conventions
    - More registers (can pass parameters too), fewer **`pushes`**/**`pops`**
  - Result? Fewer complicated instructions
    - Moved from ~1.6 μops / x86 insn to ~1.1 μops / x86 insn
- More recent: "macro-op fusion" and "micro-op fusion"
  - Intel's recent processors fuse certain instruction pairs
  - Macro-op fusion: fuses "compare" and "branch" instructions
  - Micro-op fusion: fuses load/add pairs, fuses store "address" & "data"

Post-RISC: VLIW and EPIC

- ISAs explicitly targeted for multiple-issue (superscalar) cores
  - VLIW: Very Long Insn Word
  - Later rebranded as "EPIC": Explicitly Parallel Insn Computing

- Intel/HP IA64 (Itanium): 2000
  - EPIC: 128-bit 3-operation bundles
  - 128 64-bit registers
  + Some neat features: Full predication, explicit cache control
    - Predication: every instruction is conditional (to avoid branches)
  – But lots of difficult to use baggage as well: software speculation
    - Every new ISA feature suggested in last two decades
  – Relies on younger (less mature) compiler technology
  – Commercially dead – last Itanium in 2017

ISA Research

- Compatibility killed ISA research for a while
  - But binary translation/emulation has revived it
  - … and so did the end of moore's law

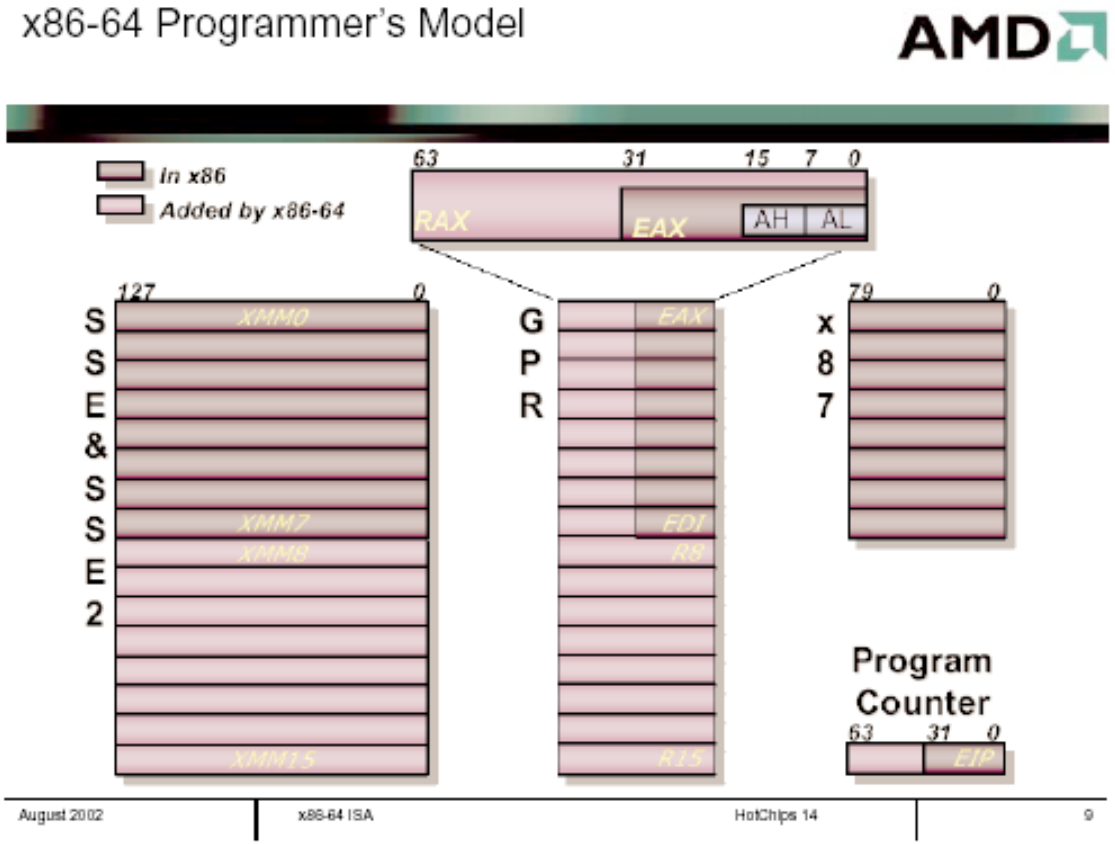
- Some ISA-related projects
  - General Purpose Dataflow
    - "WaveScalar" [Washington], "TRIPS EDGE" [Texas]
      - Explicit dataflow ISAs (vonNeumann alternatives)
    - DySER [Wisconsin], CCA,BERET[Michigan]
      - (arguably) Dataflow extensions to VonNeumann ISA
  - Specialized computers:
    - Stream-Dataflow [me]
      - Explicit memory movement + reconfigurable CGRA

x86: Registers

- 4 arithmetic,
- 4 address,
- 4 segment,
- 2 control

blech!

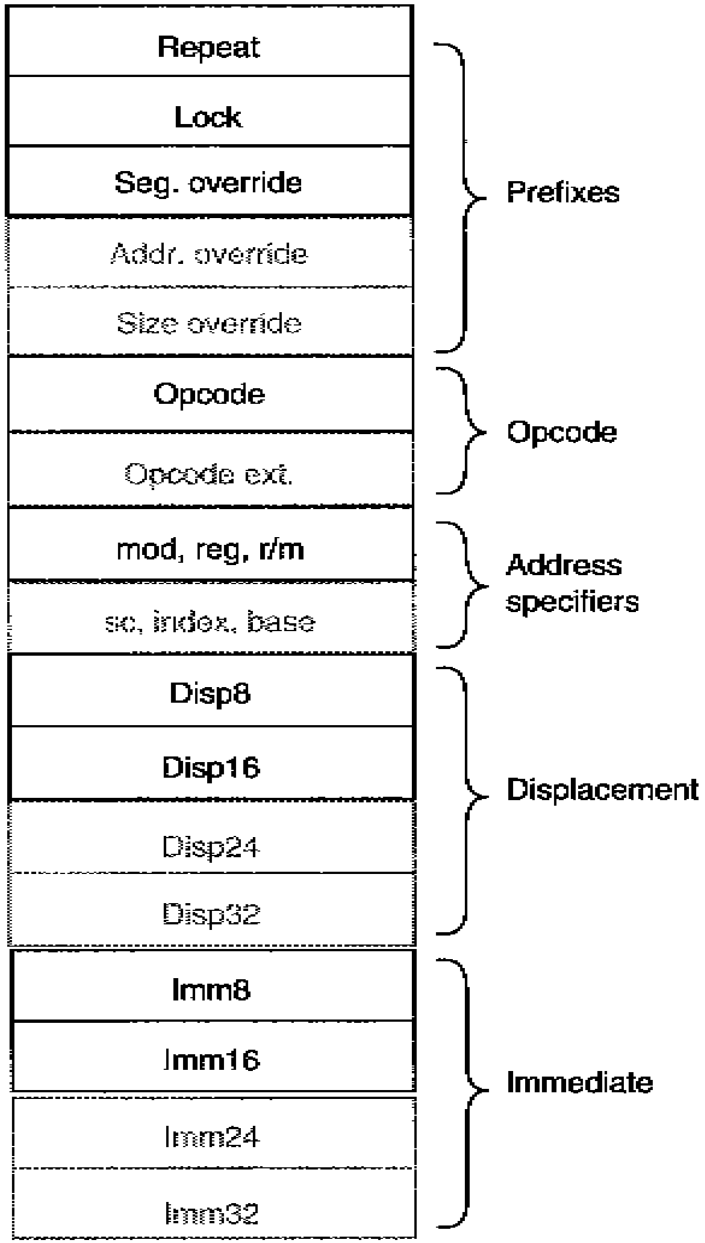
- Accumulator
  - AH, AL (8 bits)
  - AX (16 bits)
  - EAX (32 bits)
  - RAX (64 bits)



x86 Details

- Seven address modes
  - Absolute
  - Register indirect
  - Based
  - Indexed
  - Based indexed with displacement
  - Based with scaled indexed
  - Based with scaled indexed and displacement

yak!

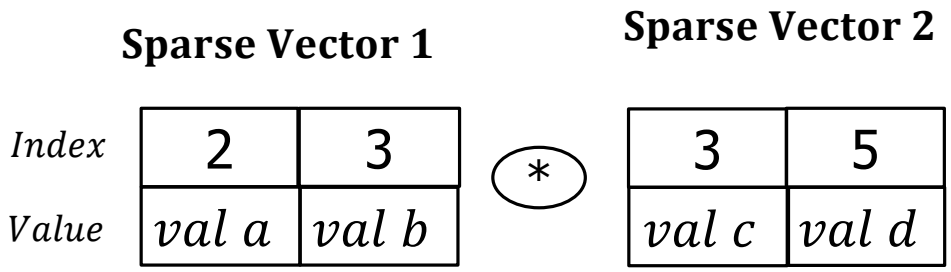


Many instruction formats

Example Program: Sparse Vector Multiply

```
typedef struct node {
  int index;
  float val;
} node;

float func(node* n1, node* n2)
{
  float total=0;
  while(n1 && n2) {
    if(n1->index==n2->index) {
      total+=n1->val*n2->val;
    } else if (n1->index >
                n2->index) {
      n1++;
    } else {
      n2++;
    }
  }
}
```



Match with 3 -> Perform $val\ b\ *\ val\ c$

Von Neumann VS. Dataflow Program

Von Neuman

```
101b0: beqz a0,101c8 <func+0x2e>
101b2: beqz a1,101c8 <func+0x2e>
101b4: lw a4,0(a0)
101b6: lw a5,0(a1)
101b8: beq a4,a5,101a0 <func+0x6>
101a0: flw fa5,4(a0)
101a4: flw fa4,4(a1)
101a8: fmul.s fa5,fa5,fa4
101ac: fadd.s fa0,fa0,fa5
101bc: ble a4,a5,101c4 <func+0x2a>
101c0: addi a0,a0,8
101c2: j 101b0 <func+0x16>
101c4: addi a1,a1,8
101c6: j 101b0 <func+0x16>
```

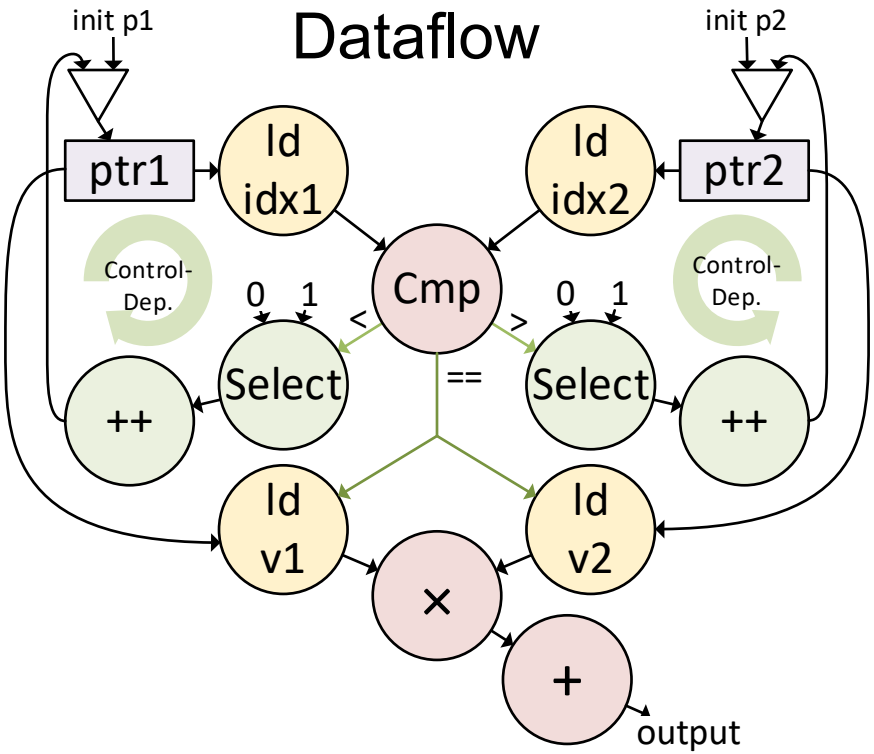
Exit

Ind1==ind2

Ind1<ind2

Ind2<ind1

Dataflow



Sparse Vector 1

Index	2	3
Value	*val a*	*val b*

*

Sparse Vector 2

3	5
val c	*val d*

Match index 3 -> Perform $val\ b\ * val\ c$

Virtual ISAs

- Move portability concerns to the software level!
- Java and C# use an ISA-like interface
  - JavaVM uses a stack-based bytecode
  - C# has the CLR (common language runtime)
  - NVIDIA's "PTX"
  - Higher-level than machine ISA
    - Design for translation (not direct execution)
- Goals:
  - Portability (abstract away the actual hardware)
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Flexibility over time

- May allow ISA research to overcome compatibility "gorilla"
  - But Intel wants x86 to be the winning "virtual ISA"

Transmeta's Take: Code Morphing

- **Code morphing**: x86 translation performed in software
  - Crusoe/Astro are x86 emulators, no actual x86 hardware anywhere
  - Only "code morphing" translation software written in native ISA
  - Native ISA is invisible to applications, even BIOS
  - Different Crusoe versions have (slightly) different ISAs: can't tell
- How was it done?
  - Code morphing software resides in boot ROM
  - On startup boot ROM hijacks 16MB of main memory
  - Translator loaded into 512KB, rest is **translation cache**
  - Software starts running in **interpreter** mode
  - Interpreter profiles to find "hot" regions: procedures, loops
  - Hot region compiled to native, optimized, cached
  - Gradually, more and more of application starts running native

Translation and Virtual ISAs

- New compatibility interface: ISA + translation software
  - Less necessary that processor ISA be compatible
  - As long as some combination of ISA + software translation layer is
- Kinds
  - **Binary-translation**: transform static image, run native
  - **Emulation**: unmodified image, interpret each dynamic insn
    - Typically optimized with just-in-time (JIT) compilation
- Examples: FX!32 (x86 on Alpha), Rosetta (PowerPC on x86)
- Downside: performance overheads
  - Performance overheads reasonable (many recent advances)

Adventures in ISA Misteps (funny as in *uh-oh*)

Motivation: Function calls incur many overhead instructions to save/restore registers.

Idea: Specialize communication between functions by exposing function->function dataflow to ISA?

Register Windows

- **Register windows**: hardware activation records
  - Sun SPARC (from the RISC I)
  - 32 integer registers divided into: 8 global, 8 local, 8 input, 8 output
  - Explicit **`save`**/**`restore`** instructions
    - Global registers fixed
    - **`save`**: inputs "pushed", outputs → inputs, locals zeroed
    - **`restore`**: locals zeroed, inputs → outputs, inputs "popped"
    - Hardware stack provides few (8) on-chip register frames
    - Spilled-to/filled-from memory on over/under flow
  - \+ Automatic parameter passing, caller-saved registers
  - \+ No memory traffic on shallow (<8 deep) call graphs
  - – Hidden memory operations (some restores fast, others slow)
  - – A nightmare for register renaming (more later)

Motivation: Pipelined processor stalls after branch instructions because it takes time to resolve the branch.

Idea: Play with program semantics to eliminate?

Delay Slot

```
ble  a4,a5,101c4
addi a0,a0,8
j      101b0
```


This instruction always executes

- **Branch Delay Slot**:
  - Instrution after branch always executes
  - Possible to have multiple! (or have load delay slots)
  - Example ISAs: MIPS, PA-RISC, ETRAX CRIS, SuperH, and SPARC
- Good
  - Helps prevent pipeline bubbles/hazards (more on that later)
- Bad
  - Inelegant: Exposes the pipelined nature of the h/w to the ISA
  - Totally irrelevant for OOO execution – headache to deal with
  - Extra compiler/assembler work
- Later RISC ISAs do not include: PowerPC, ARM, Alpha, and RISC-V