CSE 548: Computer Systems Architecture

Instruction Set Architectures

Luis Ceze, Spring 2017

(based on slides lifted from friends at UPenn, UIUC, UW, MIT, etc. but mostly from Milo Martin at UPenn)

Announcements

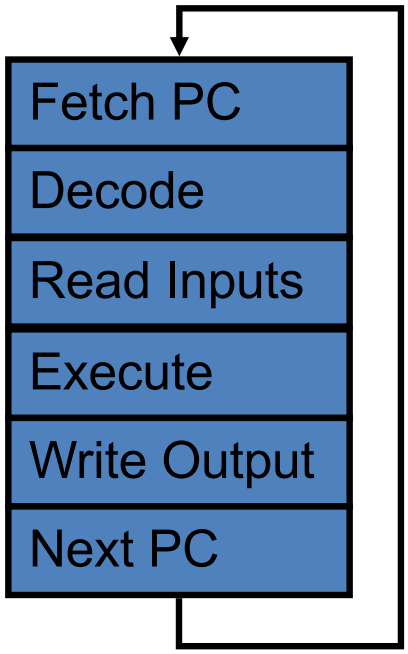
- *Power: A First-Class Architectural Design Constraint*
  - Critique due today
- Projects
  - Ideas? Need a project partner?
  - Sign up to talk to us!
- Next two lectures:
  - Workshop on Approximate Computing
  - Enabling In-network Computation with a Programmable Network Middlebox
  - Approximate Storage for Encrypted and Compressed Videos

What Is An ISA?

What Is An ISA?

- **ISA (instruction set architecture)**
  - A precisely-defined hardware/software interface, a **"contract"**
    - **Functional definition** of
      - operations, modes, and storage locations supported by hardware
    - **Description** of how to invoke operations
    - defines the *software-visible state* of the system *(what is part of this state?)*
    - defines how each instruction changes that state
    - defines instructions and encodings
  - Not in the "contract": non-functional aspects
    - How operations are implemented
    - Which operations are fast and which are slow and when
    - Which operations take more power and which take less
- *Why separate architecture and implementation*?

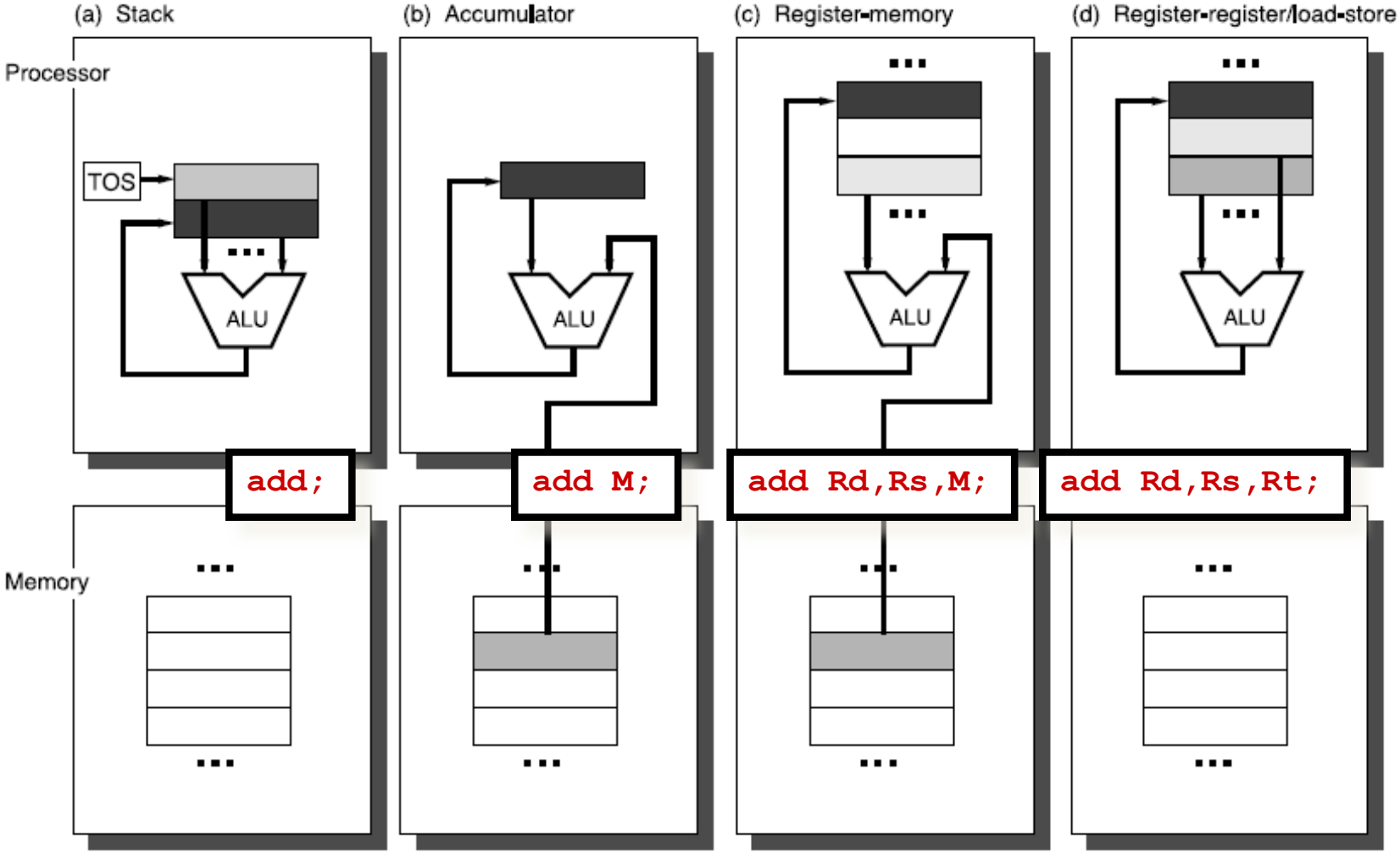
The Base Sequential Model



- Implicit model of all modern ISAs
  - Often called VonNeuman, but in ENIAC before
- Basic feature: the **program counter (PC)**
  - Defines **total order** on dynamic instruction
    - Next PC is PC++ unless insn says otherwise
  - Insn order and **named storage** define computation
    - Value flows from insn X to Y via storage A iff…
    - X names A as output, Y names A as input…
    - And Y after X in total order
- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before insn X+1 starts

Classifying ISAs (Operand models)

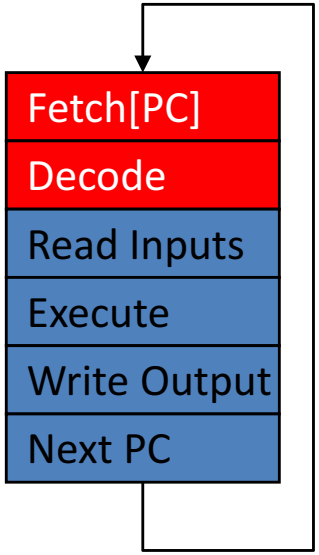
- Basic differentiation: type of internal storage



Examples of Operand Models

- ARM
  - Integer: 32 32-bit general-purpose registers (load/store)
  - Floating point: same (can also be used as 16 64-bit registers)
  - 16-bit displacement addressing
- x86
  - Integer: 8 accumulator registers (reg-reg, reg-mem, mem-reg)
    - Can be used as 8/16/32 bits
  - Floating point: 80-bit **stack** (why x86 had slow floating point)
  - Displacement, absolute, reg indirect, indexed and scaled addressing
    - All with 8/16/32 bit constants (why not?)
  - Note: integer **`push`**, **`pop`** for managing software stack
  - Note: also reg-mem and mem-mem string functions in hardware
- x86-64 (i.e., IA32-EM64T)
  - Integer: **16 64-bit accumulator registers**
  - Floating point: **16 128-bit accumulator registers**

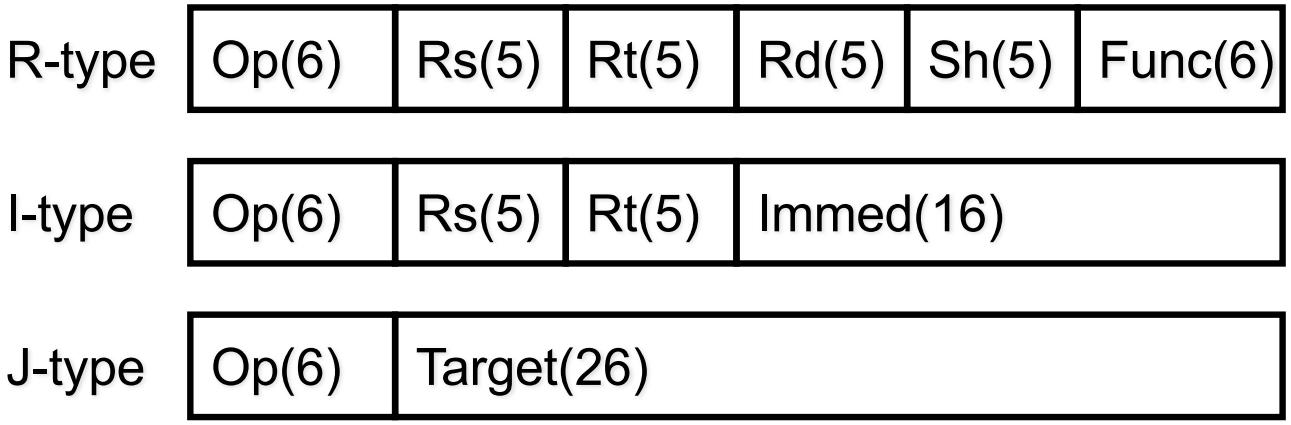
Instruction Length and Format



- **Length**
  - Fixed length
    - Most common is 32 bits
    - \+ Simple implementation (next PC often just PC+4)
    - – Code density: 32 bits to increment a register by 1
  - Variable length
    - \+ Code density
      - – x86 can do increment in one 8-bit instruction
    - – Complex fetch (where does next instruction begin?)
  - Compromise: two lengths
    - E.g., MIPS16 or ARM's Thumb
- **Encoding**
  - A few simple encodings simplify decoder
    - x86 decoder one of nastiest pieces of logic

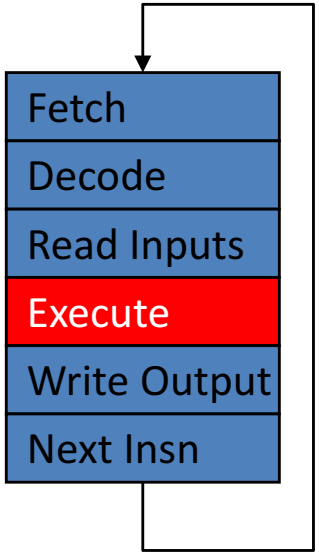
Examples Instruction Encodings

- MIPS
  - Fixed length
  - 32-bits, 3 formats, simple encoding
  - (MIPS16 has 16-bit versions of common insn for code density)



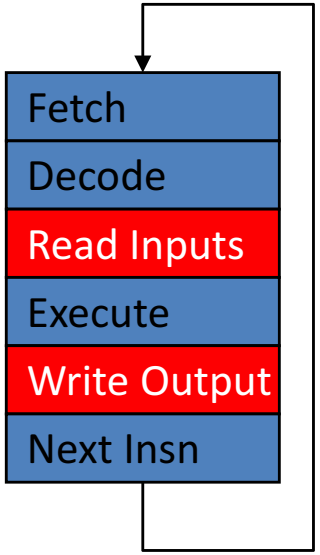
- x86
  - Variable length encoding (1 to 16 bytes)

Operations and Datatypes



- Datatypes
  - Software: attribute of data
  - Hardware: attribute of operation, data is just 0/1's
- All processors support
  - 2C integer arithmetic/logic (8/16/32/64-bit)
  - IEEE754 floating-point arithmetic (32/64 bit)
    - Intel has 80-bit floating-point
- More recently, most processors support
  - "Packed-integer" insns, e.g., MMX
  - "Packed-fp" insns, e.g., SSE/SSE2
  - For multimedia, more about these later

Where Does Data Live?



- **Memory**
  - Fundamental storage space
- **Registers**
  - Faster than memory, quite handy
  - Most processors have these too
- Immediates
  - Values spelled out as bits in instructions,
    - What does this imply?
  - Input only
    - Why?

How Much Memory? Address Size

- What does “64-bit” in a 64-bit ISA mean?

How Much Memory? Address Size

- What does "64-bit" in a 64-bit ISA mean?
  - **Support memory size of 2^{64}**
  - Alternative definition: width of calculation operations
- **Virtual address size**
  - Determines size of addressable (usable) memory
    - Current 32-bit or 64-bit address spaces
    - All ISAs moving to (if not already at) 64 bits
  - Most critical, inescapable ISA design decision
    - Too small? Will limit the lifetime of ISA
    - May require nasty hacks to overcome (E.g., x86 segments)
  - x86 evolution:
    - 4-bit (4004), 8-bit (8008), 16-bit (8086), 24-bit (80286),
    - 32-bit + protected memory (80386)
    - 64-bit (AMD's Opteron & Intel's EM64T Pentium4)

How Many Registers?

- Registers faster than memory, have as many as possible?
- What can we put in registers?
- How are they addressed?

How Many Registers?

- Registers faster than memory, have as many as possible?
  - **No**
  - One reason registers are faster is that there are **fewer of them**
    - Small is fast (hardware truism)
  - 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**
  - Upshot: trend to more registers: 8 (x86)→32 (MIPS) →128 (IA64)
    - 64-bit x86 has 16 64-bit integer and 16 128-bit FP registers

Registers vs Memory

- What is the *fundamental* difference in how they are used?

How Are Locations Specified?

- Registers are specified **directly as immediates**
  - Register names are short, can be encoded in instructions
  - Some instructions implicitly read/write certain registers

- How are addresses specified? As variables/expressions
  - Addresses are long (64-bit)
  - **Addressing mode**: how are insn bits converted to addresses?
  - Think about: what high-level idiom addressing mode captures

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 (Annoying) Addressing Issues

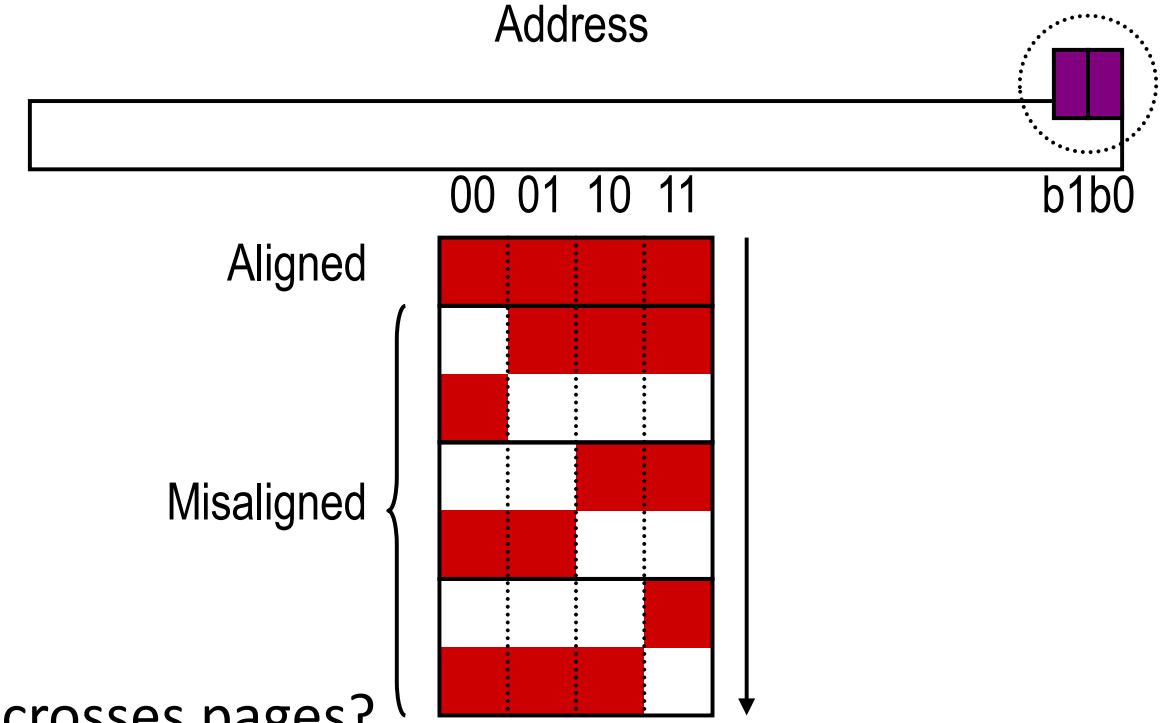
- **Access alignment**: address % size == 0?
  - Aligned: **`load-word @XXXX00, load-half @XXXXX0`**
  - Unaligned: **`load-word @XXXX10, load-half @XXXXX1`**
  - Question: what to do with unaligned accesses (uncommon case)?
    - Support in hardware? Makes all accesses slow
    - Trap to software routine? Possibility
    - Use regular instructions
      - Load, shift, load, shift, and
    - **MIPS? ISA support**: unaligned access using two instructions

      `lwl @XXXX10; lwr @XXXX10`

- **Endian-ness**: arrangement of bytes in a word
  - Why little endian? To be different? To be annoying? Nobody knows

Why alignment matters: Example

- 32-bit word: one or two accesses?

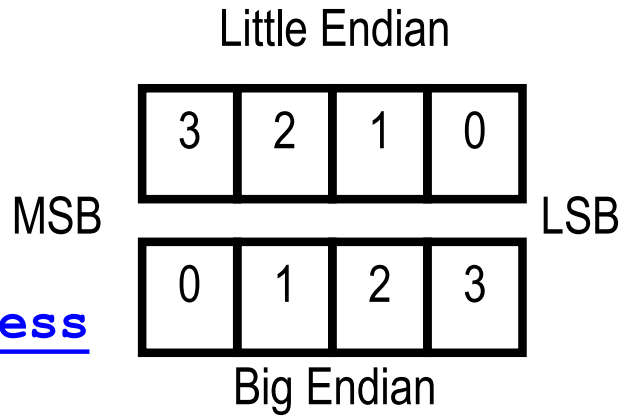


- What if it crosses pages?
- Some architecture have ***alignment exceptions!***
- Aligned architecture can make *b1b0* implicit
  - important trick for instruction encoding too, *why?*

Byte Ordering

- Two Conventions
  - Big Endian, specify address of most significant byte
  - Little Endian, specify address of least significant byte
- No technical significance to distinction – just religious!
  - Big Endian: Amiga, Macintosh, IBM RS6000, SGI, Sun
  - Little Endian: DEC, IBM PC
  - recently many processors are "bimodal"
    - MIPS, PowerPC (both mostly Big Endian)
- Names based on Gulliver's Travels

  http://www.wikipedia.org/wiki/Endianness



Which one do you prefer? Why?

ISAs Also Include Support For…

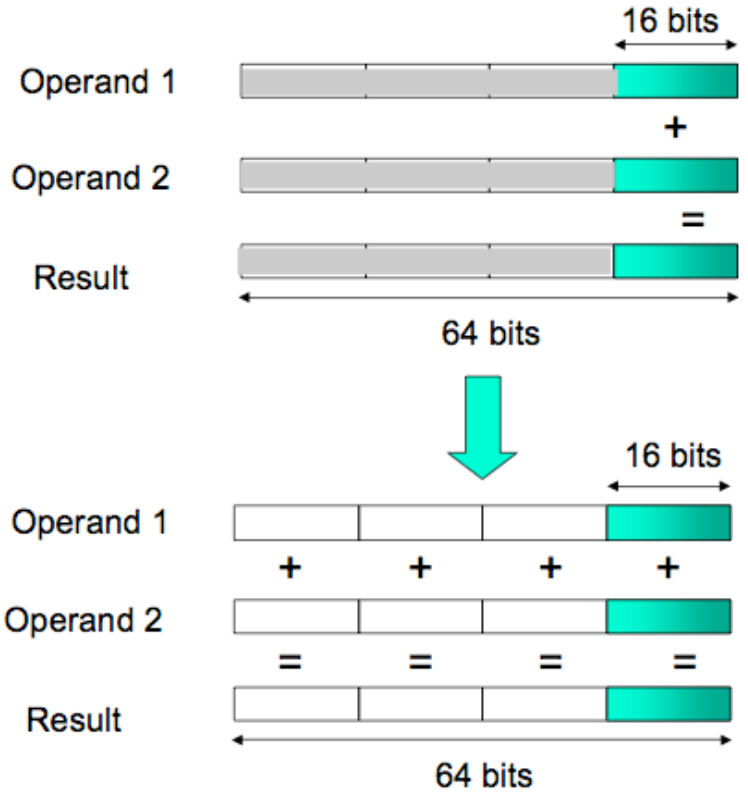
- Function calling conventions
  - Which registers are saved across calls, how parameters are passed
- Operating systems & memory protection
  - Privileged mode
  - System call (TRAP)
  - Exceptions & interrupts
  - Interacting with I/O devices
- Multiprocessor support
  - "Atomic" operations for synchronization
- Data-level parallelism
  - Pack many values into a wide register
    - Intel's SSE2: four 32-bit float-point values into 128-bit register
  - Define parallel operations (four "adds" in one cycle)

Data-Level Parallel ISA Extensions

Aka, “multimedia” instructions.

- Multiple favors
  - ia32 family: MMX, SSE, SSE2
  - PowerPC: Altivec
  - sparc: VIS

- ***What can we use them for?***



ISA Implementability

- Every ISA can be implemented
  - Not every ISA can be implemented efficiently
- Classic high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution (more later)
- Certain ISA features make these difficult
  - Variable instruction lengths/formats: complicate decoding
  - Implicit state: complicates dynamic scheduling
  - Variable latencies: complicates scheduling
  - Difficult to interrupt instructions: complicate many things

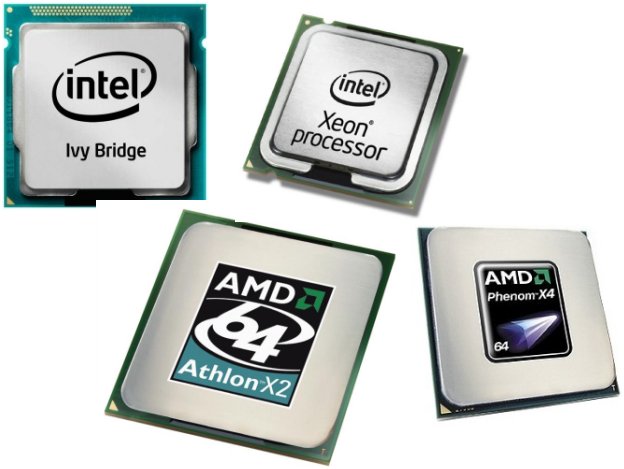
Architecture or Implementation?

- No. of GP registers
- Width of the data bus
- Binary representation of the instruction
- No. of cycles a floating point add takes
- No. of cycles processor must wait after a load before it can use the data
- Floating point format supported
- Size of the instruction cache
- No. of instructions that issue each cycle
- No. of addressing modes
- *Precise exceptions?*

ARM®

RISC

x86



CISC

RISC vs CISC in one slide

- Recall performance equation:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)

- **CISC** (Complex Instruction Set Computing)
  - Reduce "instructions/program" with "complex" instructions
    - But…?
  - Easy for assembly-level programmers, good code density
- **RISC** (Reduced Instruction Set Computing)
  - Increases "instruction/program", but hopefully not as much
    - Why do it then? What happens to the poor compiler?
  - What happens to cycles/instruction?
  - And cycle time?

The RISC vs. CISC 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

Current Winner (Volume): RISC

- ARM (Acorn RISC Machine → Advanced RISC Machine)
  - First ARM chip in mid-1980s (from Acorn Computer Ltd).
  - ***Billion*** of units sold yearly(>50% of all 32/64-bit CPUs)
  - Low-power and **embedded** devices (iPod, for example)
    - Significance of embedded? New ISAs easier to pull off
- 32-bit RISC ISA
  - 16 registers, PC is one of them
  - Many addressing modes, e.g., auto increment
  - Condition codes, each instruction can be conditional
- Multiple implementations
  - X-scale (design was DEC's, bought by Intel, sold to Marvel)
  - Others: Freescale (was Motorola), Texas Instruments, STMicroelectronics, Samsung, Sharp, Philips, etc.

Current Winner (Revenue): CISC

- 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"
    - x86 is most difficult ISA to implement and do it fast but…
    - Because Intel sells the most **non-embedded** processors…
    - It has the most money…
    - Which it uses to hire more and better engineers…
    - Which it uses to maintain competitive performance …
    - **And given competitive performance, compatibility wins…**
    - So Intel sells the most **non-embedded** processors…
  - AMD as a competitor keeps pressure on x86 performance

- Moore's law has helped Intel in a big way
  - Most engineering problems can be solved with more transistors

Intel's Compatibility Trick: RISC Inside

- 1993: Intel wanted out-of-order execution in Pentium Pro
  - OoO was very 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 $eax [$esp-4]
  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
    - Result: Intel implemented OoO before any RISC company
    - Also, OoO also benefits x86 more (because ISA limits compiler)
  - Idea co-opted by other x86 companies: AMD and Transmeta

More About Micro-ops

- Even better? Two forms of hardware translation
  - Hard-coded logic: fast, but complex
  - Table: slow, but "off to the side", doesn't complicate rest of machine
- x86: average 1.6 μops / x86 insn
  - Logic for common insns that translate into 1–4 μops
  - Table for rare insns that translate into 5+ μops
- x86-64: average 1.1 μops / x86 insn
  - More registers (can pass parameters too), fewer **`pushes/pops`**
  - Core2: logic for 1–2 μops, Table for 3+ μops?
- More recent: "macro-op fusion" and "micro-op fusion"
  - E.g., fuse address calculation and access
  - E.g., fuse TEST/CMP with JMP into a single conditional jump instruction
  - Intel's recent processors fuse certain instruction pairs (ARM too!)

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 (and what we will use in our assignments), the exact micro-ops are specific to each chip

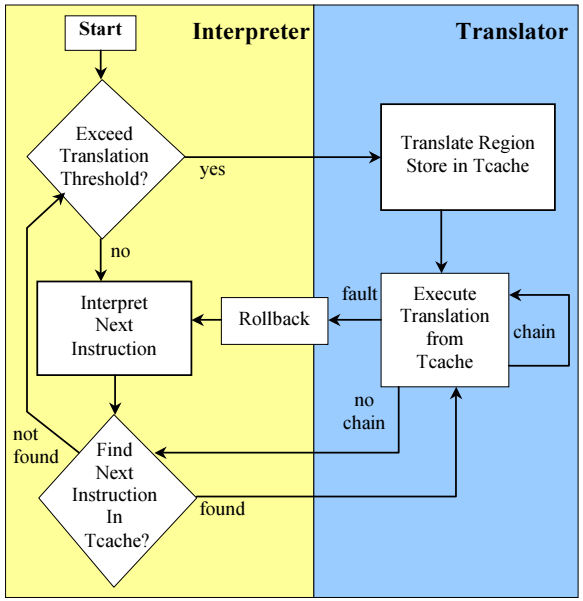
Translation and Virtual ISAs

- New compatibility interface: ISA + translation software
  - **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)
  - Performance overheads reasonable (many recent advances)

- **Virtual ISAs**: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Goals: Portability (abstract hardware nastiness), flexibility over time
  - Examples: Java Bytecodes, C# CLR (Common Language Runtime)

Transmeta's Take: Code Morphing

- **Code morphing**: x86 translation 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 and even OS
  - Different Crusoe versions have (slightly) different ISAs: can't tell
- How was it done?
  - Code morphing software resides in boot read-only memory (ROM)
  - On startup, 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



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
  - Not doing well commercially

Compiler Programmability

- What makes an ISA easy for a compiler to program in?
  - Low level primitives from which solutions can be synthesized
    - a = b*c+d
    - Computers good at breaking complex structures to simple ones
      - Requires traversal
    - Not so good at combining simple structures into complex ones
      - Requires search, pattern matching
    - Easier to synthesize complex insns than to compare them
- What do compiler optimizations do?

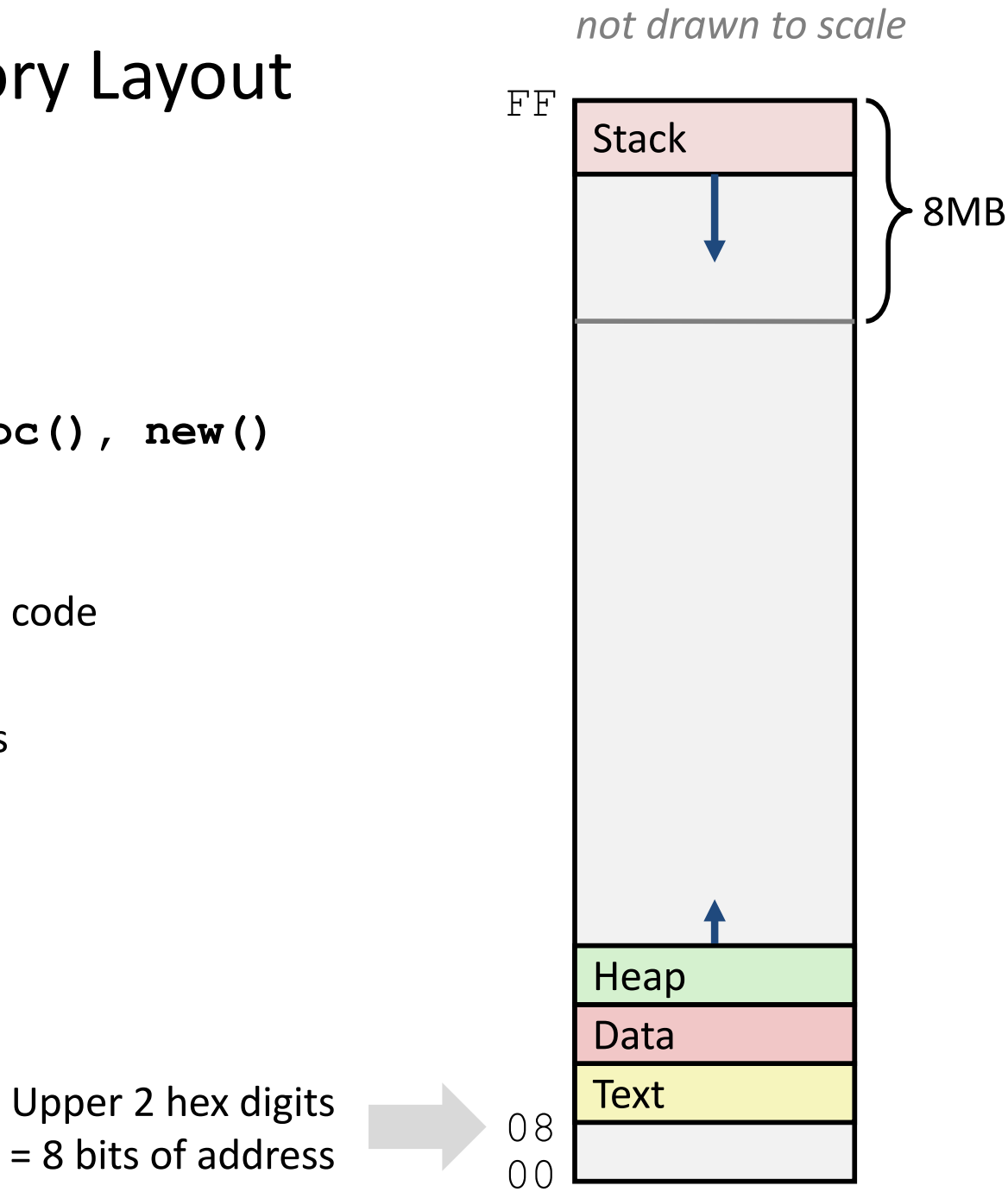
Compiler Optimizations

- Primarily reduce dynamic insn count
  - Eliminate redundant computation, keep more things in registers
    - \+ Registers are faster, fewer loads/stores
    - – An ISA can make this difficult by having too few registers
- But also…
  - Reduce branches and jumps
  - Reduce cache misses
  - Reduce dependences between nearby insns (for parallelism)
    - – An ISA can make this difficult by having implicit dependences
  - Why?
- How effective are these?
  - \+ Can give 4X performance over unoptimized code
  - – Collective wisdom of 40 years ("Proebsting's Law"): 4% per year
  - – Funny but … shouldn't leave 4X performance on the table

Quick motivating example for ISA extensions

IA32 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
- Heap
  - Dynamically allocated storage
  - When call **`malloc(), calloc(), new()`**
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only



Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

why?

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

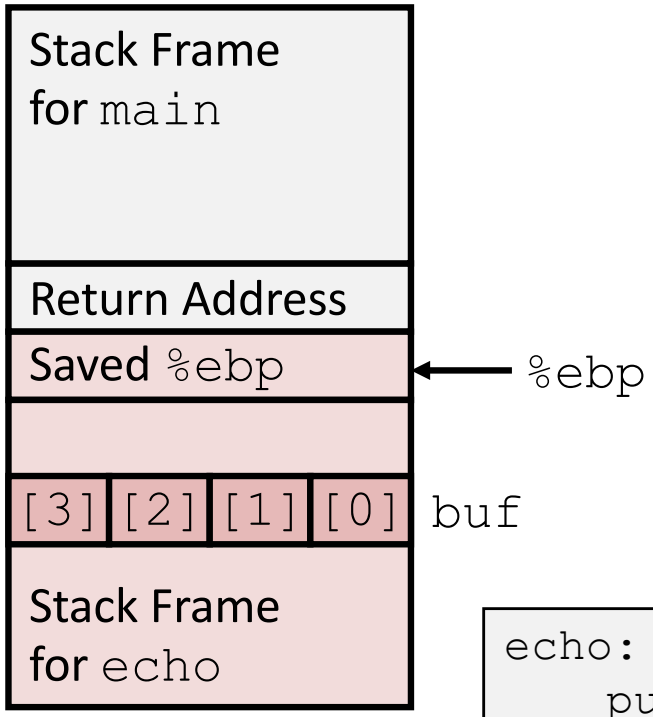
```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

Buffer Overflow Stack

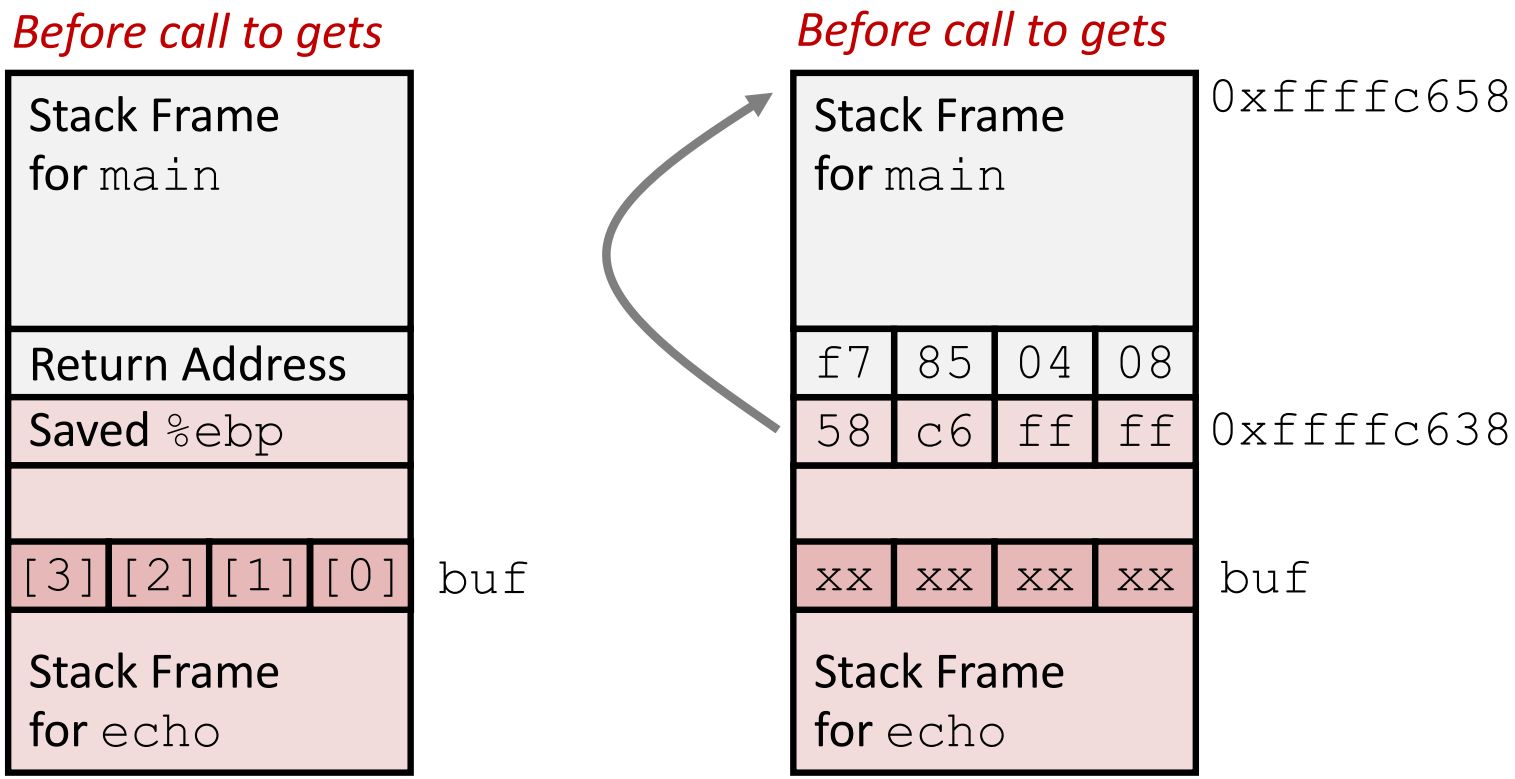
Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

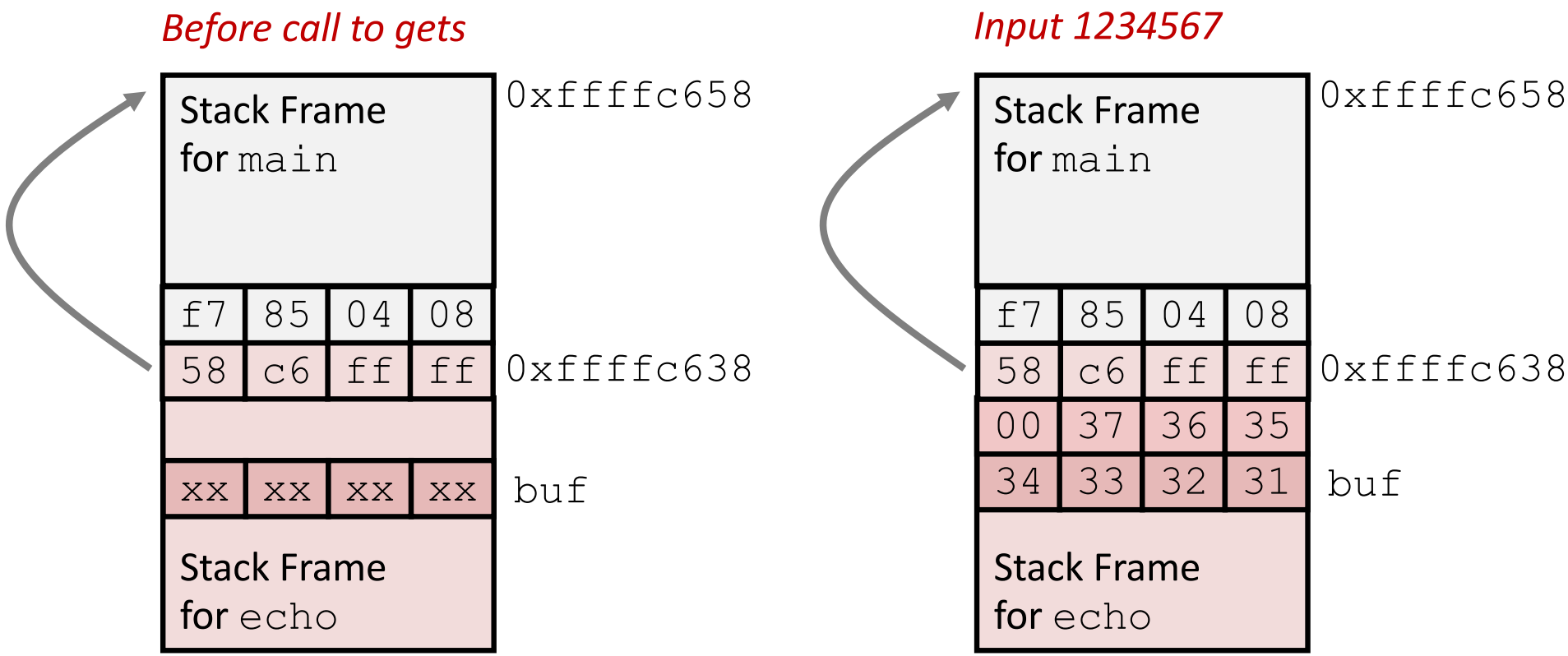
```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    subl  $20, %esp         # Allocate stack space
    movl  %ebx, (%esp)      # Push buf addr on
stack
    call  gets              # Call gets
    . . .
```

Buffer Overflow Stack Example



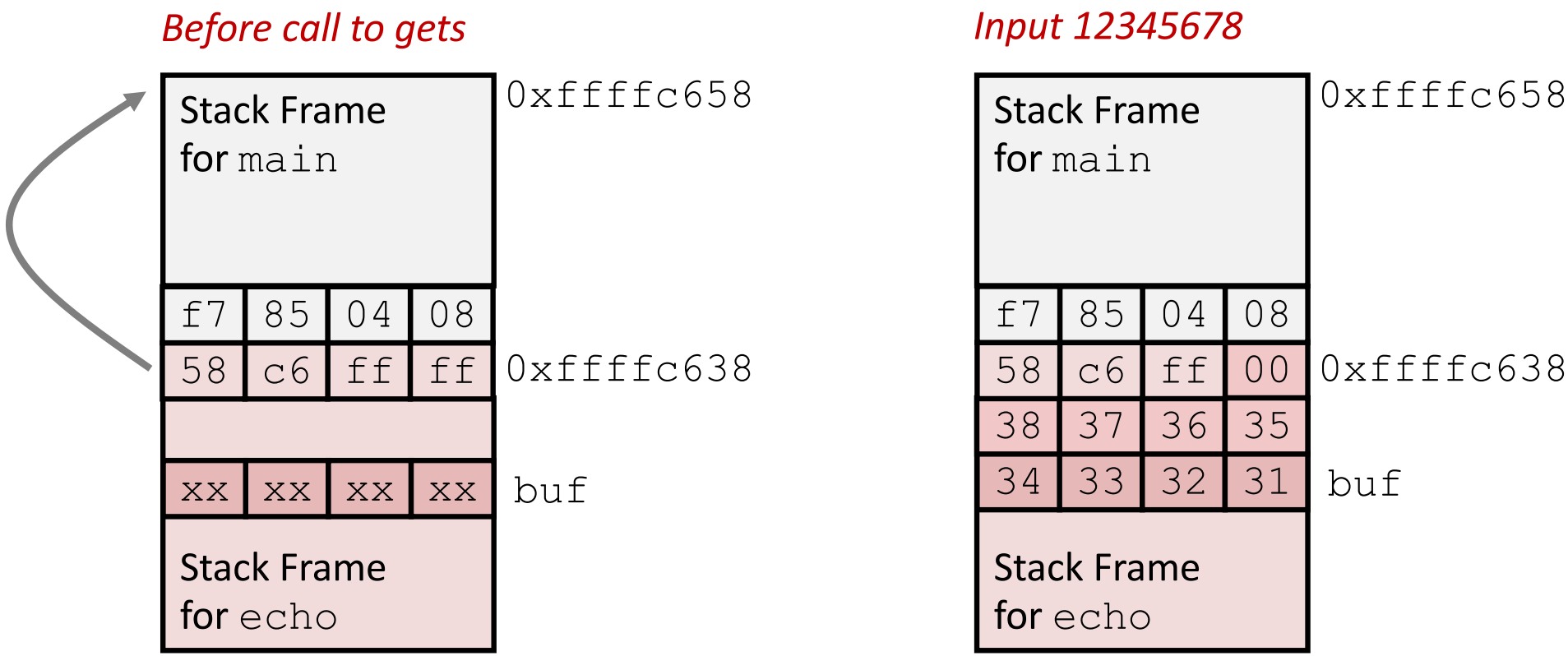

```
80485f2:call 80484f0 <echo>
80485f7:mov  0xffffffc(%ebp),%ebx # Return Point
```

Buffer Overflow Example #1



Overflow buf, but no problem

Buffer Overflow Example #2

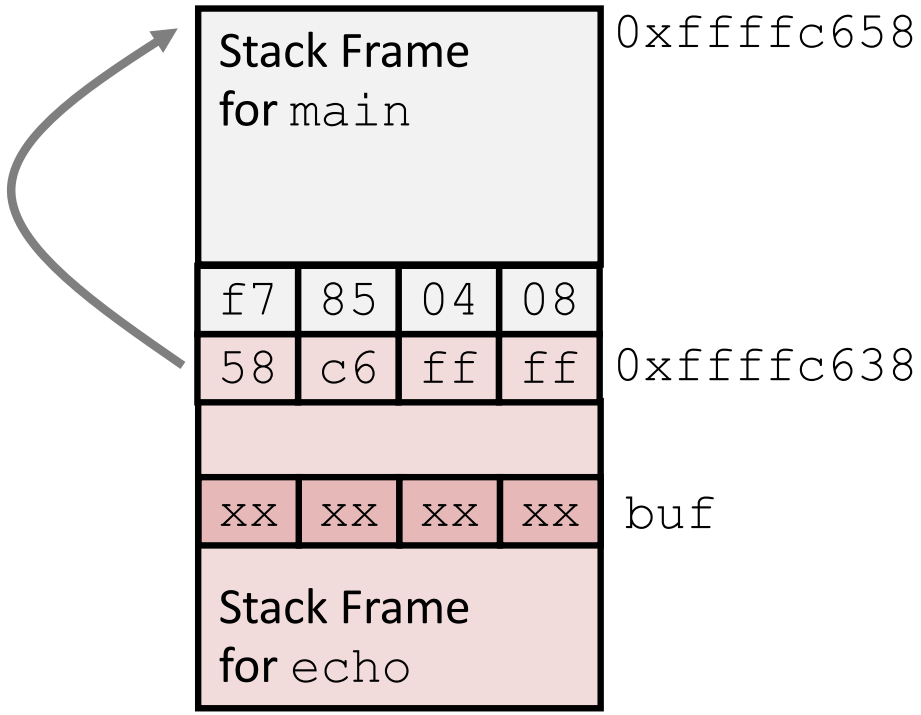


Base pointer corrupted

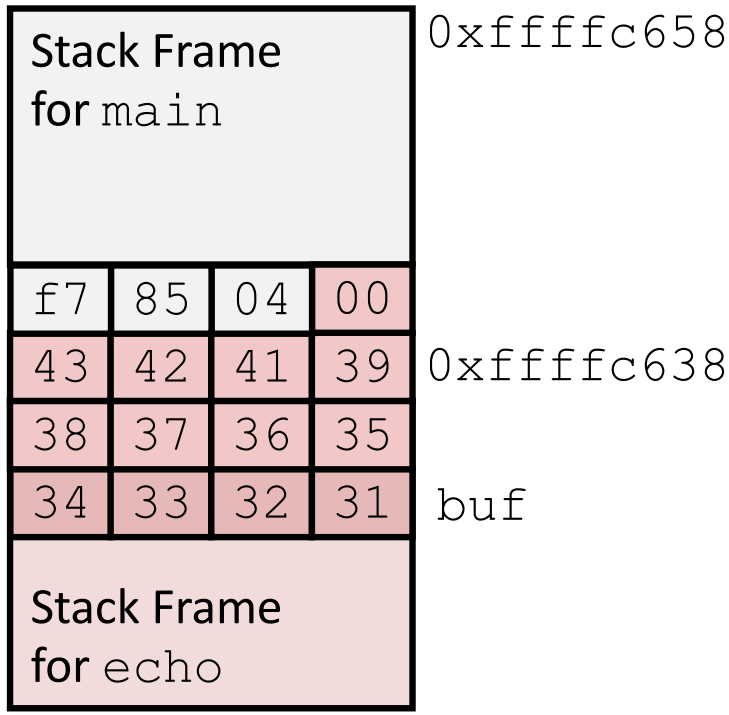
```
. . .
804850a:  83 c4 14   add    $0x14,%esp  # deallocate space
804850d:  5b         pop    %ebx        # restore %ebx
804850e:  c9         leave              # movl %ebp, %esp; popl %ebp
804850f:  c3         ret                # Return
```

Buffer Overflow Example #3

Before call to gets



Input 123456789ABC



Return address corrupted

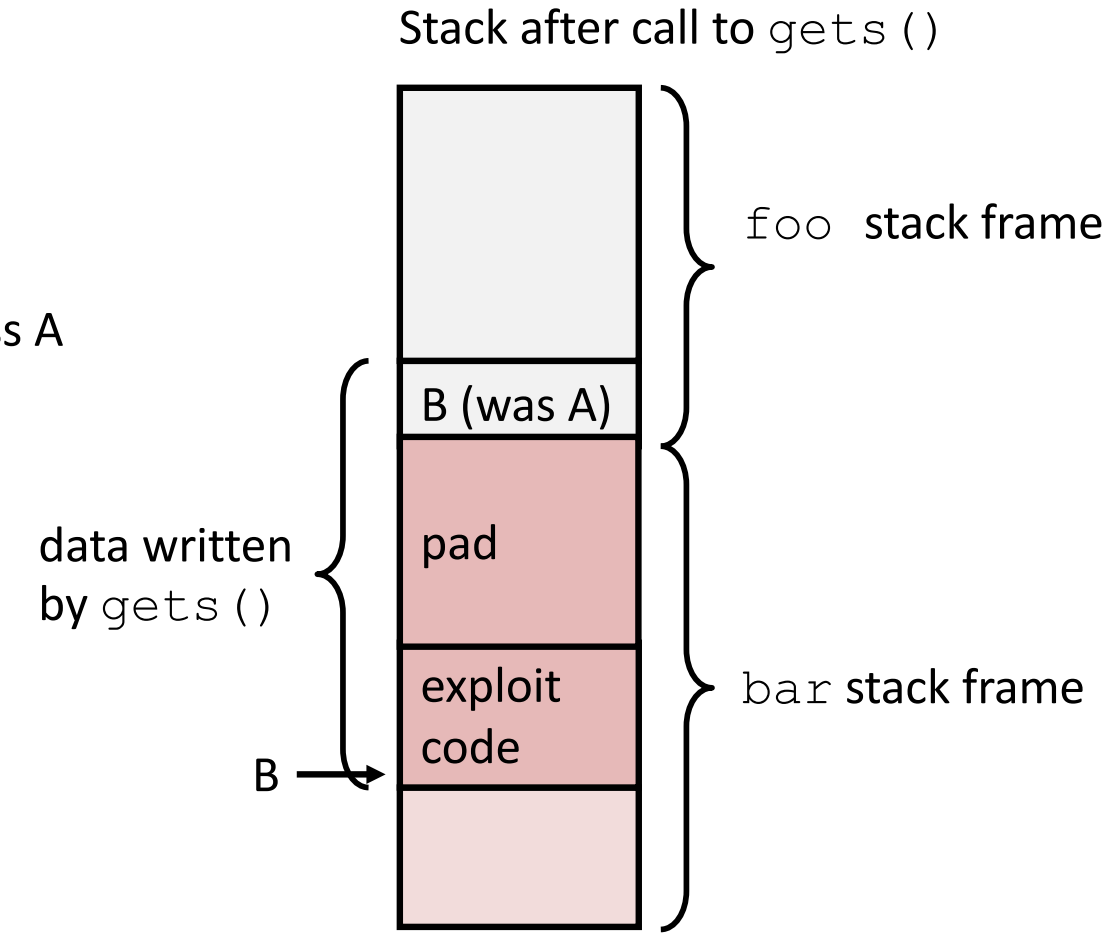
```
80485f2: call 80484f0 <echo>
80485f7: mov  0xffffffc(%ebp),%ebx # Return Point
```

Malicious Use of Buffer Overflow

```
void foo(){
  bar();
  ...
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- Input string contains byte representation of executable code
- Stack frame must be big enough to hold exploit code
- Overwrite return address with address of buffer (need to know B)
- When `bar()` executes `ret`, will jump to exploit code (instead of A)
- *What is the big problem? How can no-execute help?*

How could ISA extensions help with that problem?

ISA extensions

- Software needs often motivates new instructions
  - No execute bit for security
  - Short vectors (multimedia, games, etc)
  - fmadd (floating point multiply and add)
  - virtualization
- What would you like to see in the ISA?

- Careful: once it is in, hard to take out!
  - Backwards compatibility…

How much do ISAs really matter today?

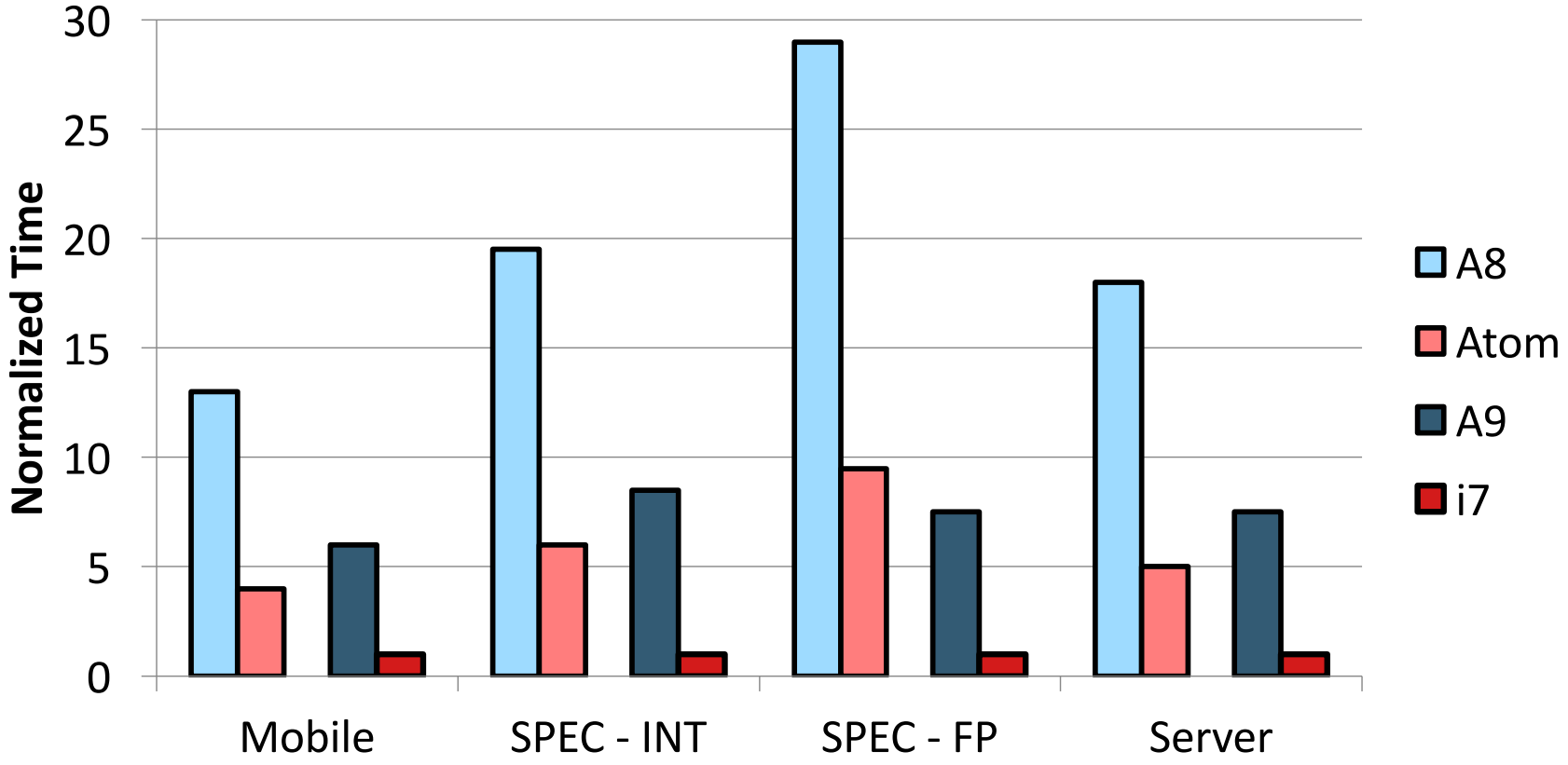
- Consider:
  - Lots of transistors available
  - Lots of code run on managed environments/virtual ISAs
- Does it matter for performance?
- Does it still matter for compatibility?

Appears in the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA 2013)

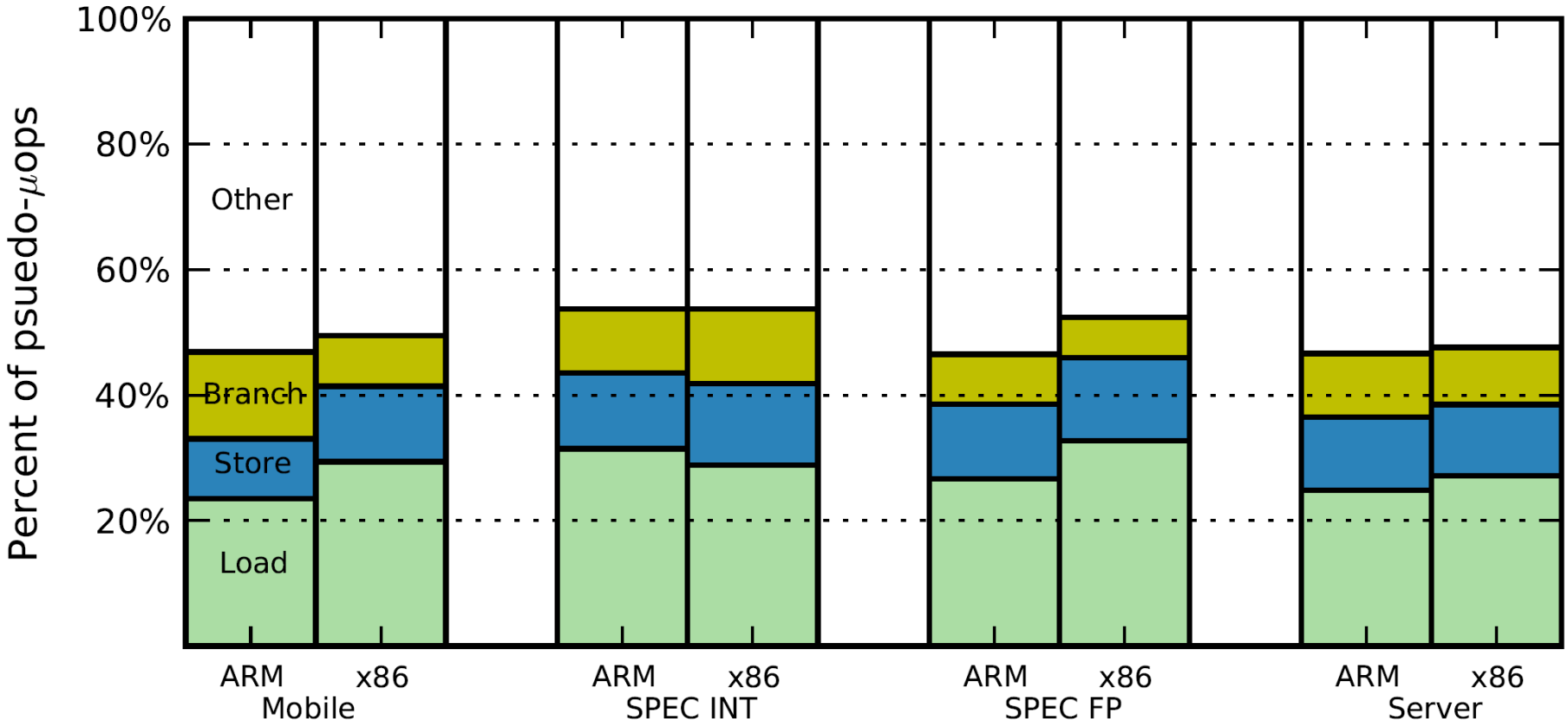
Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures

Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam
University of Wisconsin - Madison
{blem,menon,karu}@cs.wisc.edu

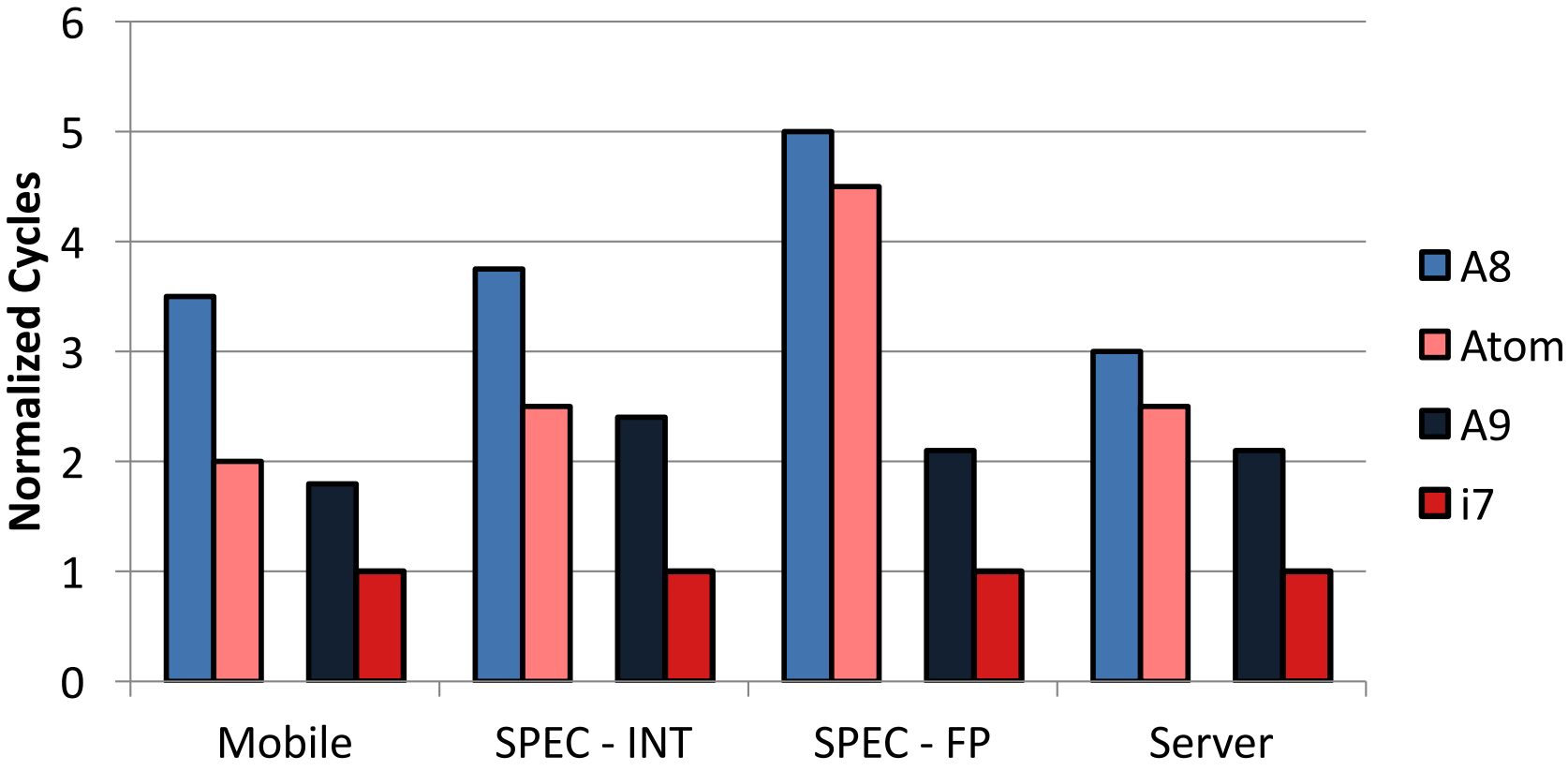
Performance



Instruction Mix



Cycle counts



But ISAs matters a lot!

Hein? Why?

But ISAs matter a lot!

- Transactional Memory support (more later)
- Dataflow (WaveScalar, TRIPS)
- Application-specific hybrid CISC/RISC (e.g., crypto)
- Approximate operations for energy (e.g., Truffle)
- Bounds-checking (e.g., HardBound)
- Information-flow tracking
- Neural networks