

18-447 Lecture 2: Development of ISAs

James C. Hoe

Department of ECE

Carnegie Mellon University

Housekeeping

- Your goal today
 - understand how ISAs got to be the way they are
- Notices
 - Lab 1, Part A, **due this week**
 - Lab 1, Part B, **due next week**
 - HW1, **due Monday 2/7 noon**
- Readings
 - P&H Ch 2 (optional P&H App D: RISC Survey)
 - optional (in supplemental handout on Canvas)
 - 1946 von Neumann paper
 - 1964 IBM 360 paper
 - P&H Ch 1.6~1.9 for next time

“ISA” in a nut shell

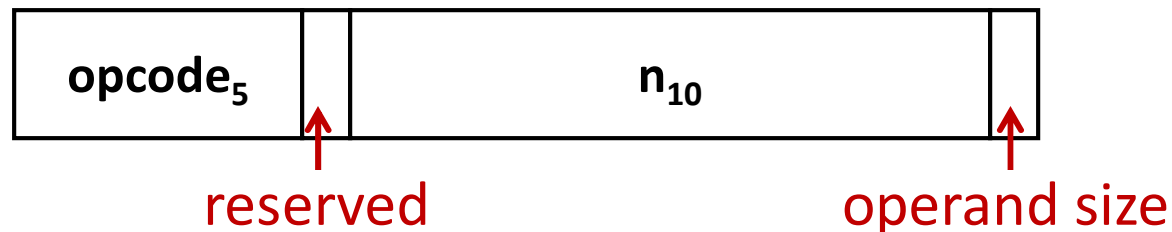
- A stable programming target (to last for decades)
 - binary compatibility for SW investments
 - permits adoption of foreseeable technology

Better to compromise immediate optimality for future scalability and compatibility

- Dominant paradigm has been “von Neumann”
 - program visible state: memory, registers, PC, etc.
 - instructions to modified state; each prescribes
 - which state elements are read
 - which state elements—including PC—updated
 - how to compute new values of update state

Atomic, sequential, in-order

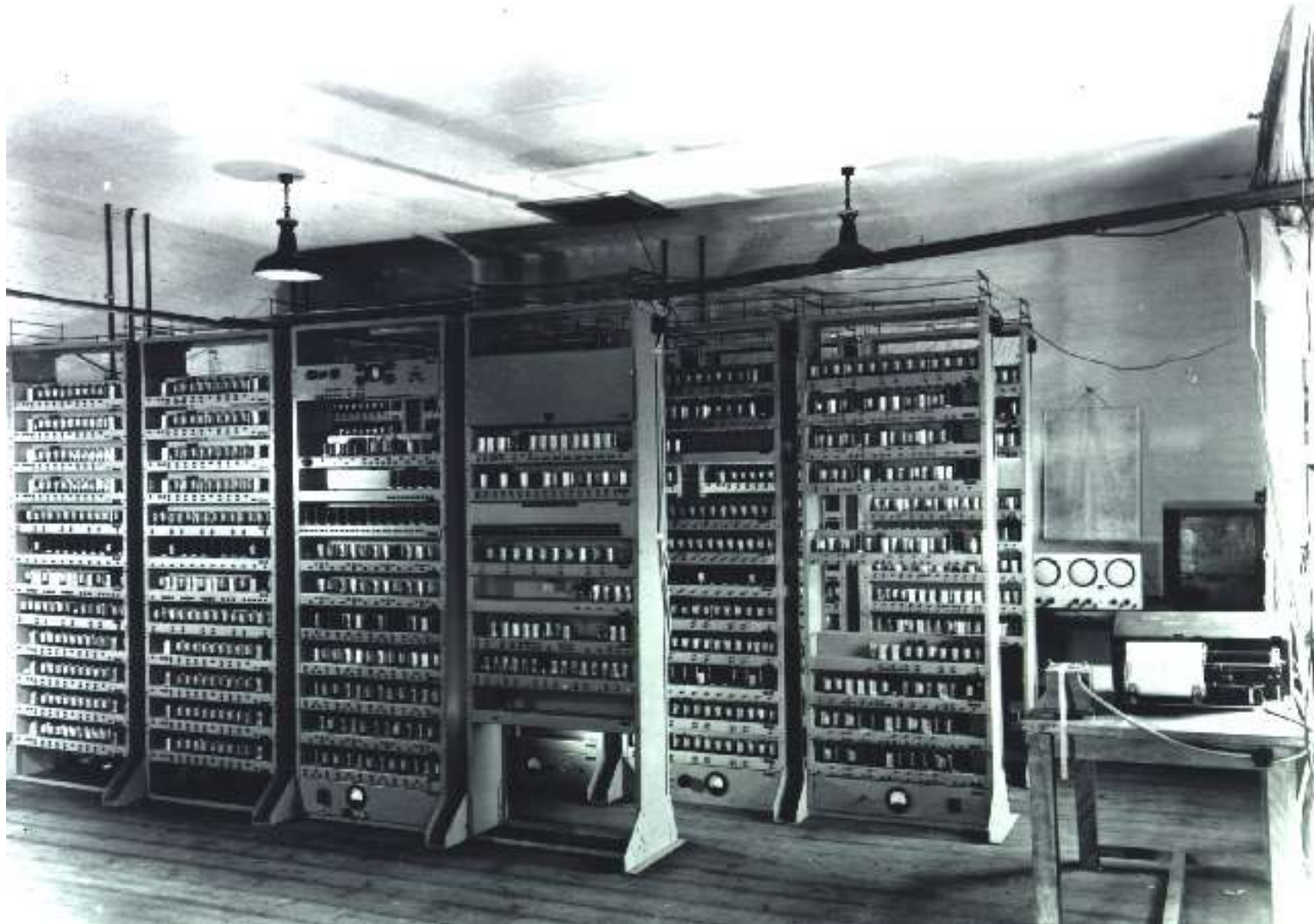
An Early ISA: EDSAC



- Single accumulator architecture, i.e. $ACC \leftarrow ACC \oplus M[n]$ (17 or 35)
- Instruction examples
 - **A n**: add $M[n]$ into ACC (also S, R, L)
 - **T n**: transfer the contents of ACC to $M[n]$ and clear
 - **E n**: If $ACC \geq 0$, branch to $M[n]$ or proceed serially
 - **I n**: Read the next character from paper tape, and store it as the least significant 5 bits of $M[n]$
 - **Z**: Stop the machine and ring the warning bell

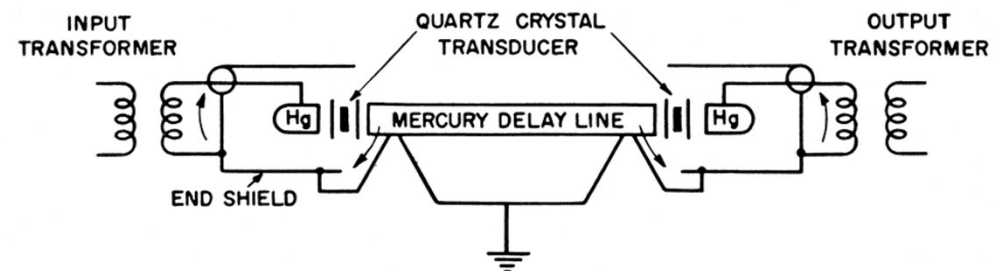
Notice: all addresses are “fixed” into instructions

Why not more registers??

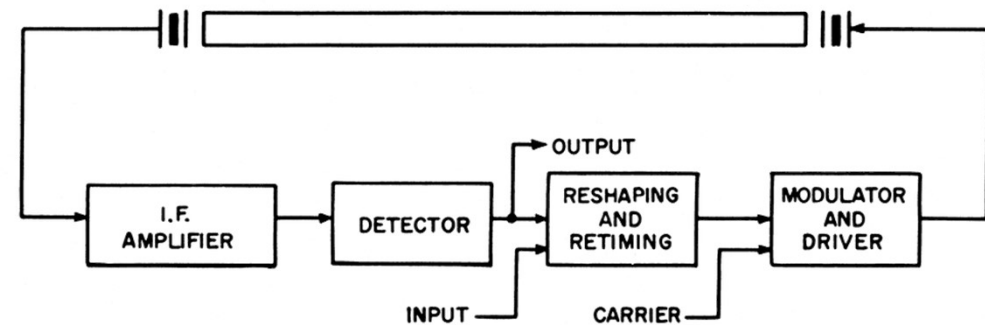


[ENIAC accumulators, Wikipedia]

BTW, this was “memory” . . .



Schematic diagram of circuit connections to the acoustic delay line used in NBS mercury memory.

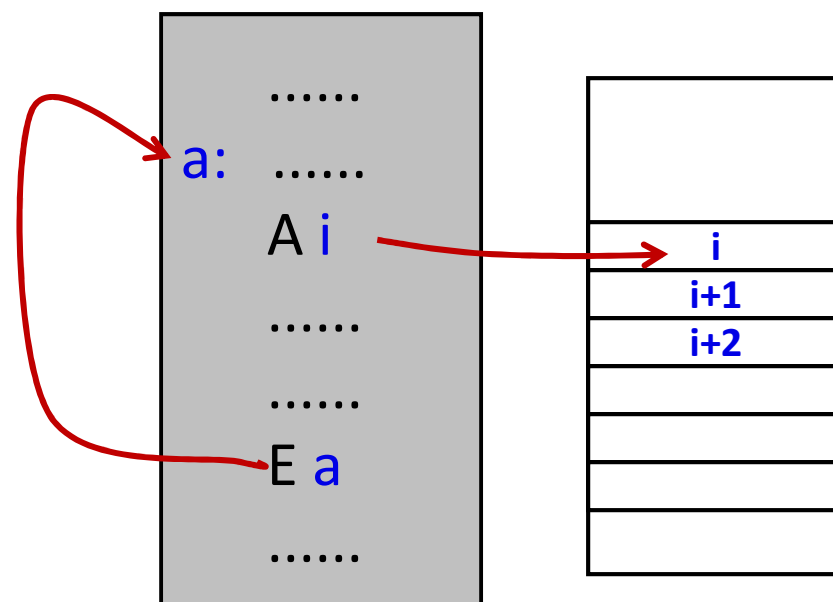
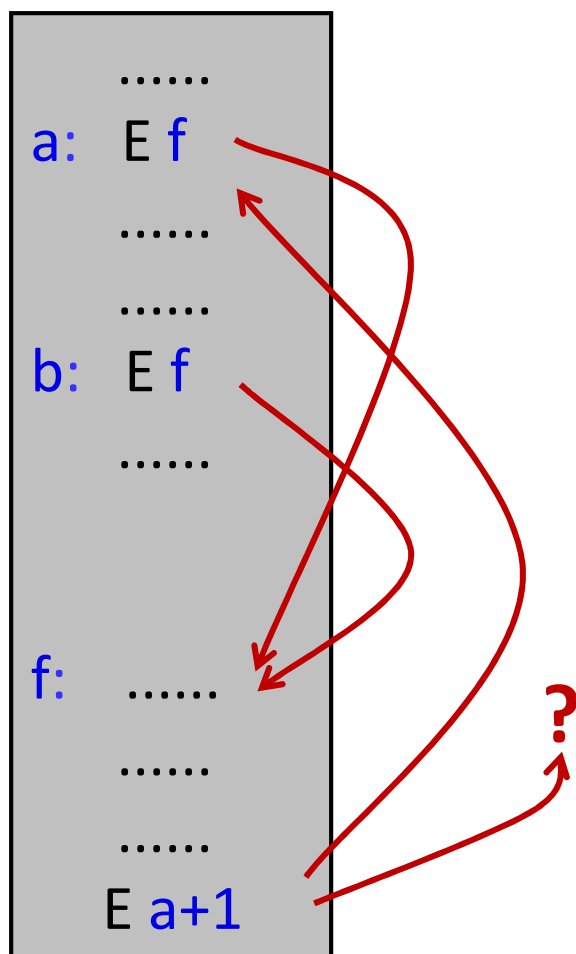


Block diagram of the mercury memory system.

[Image from Wikipedia]

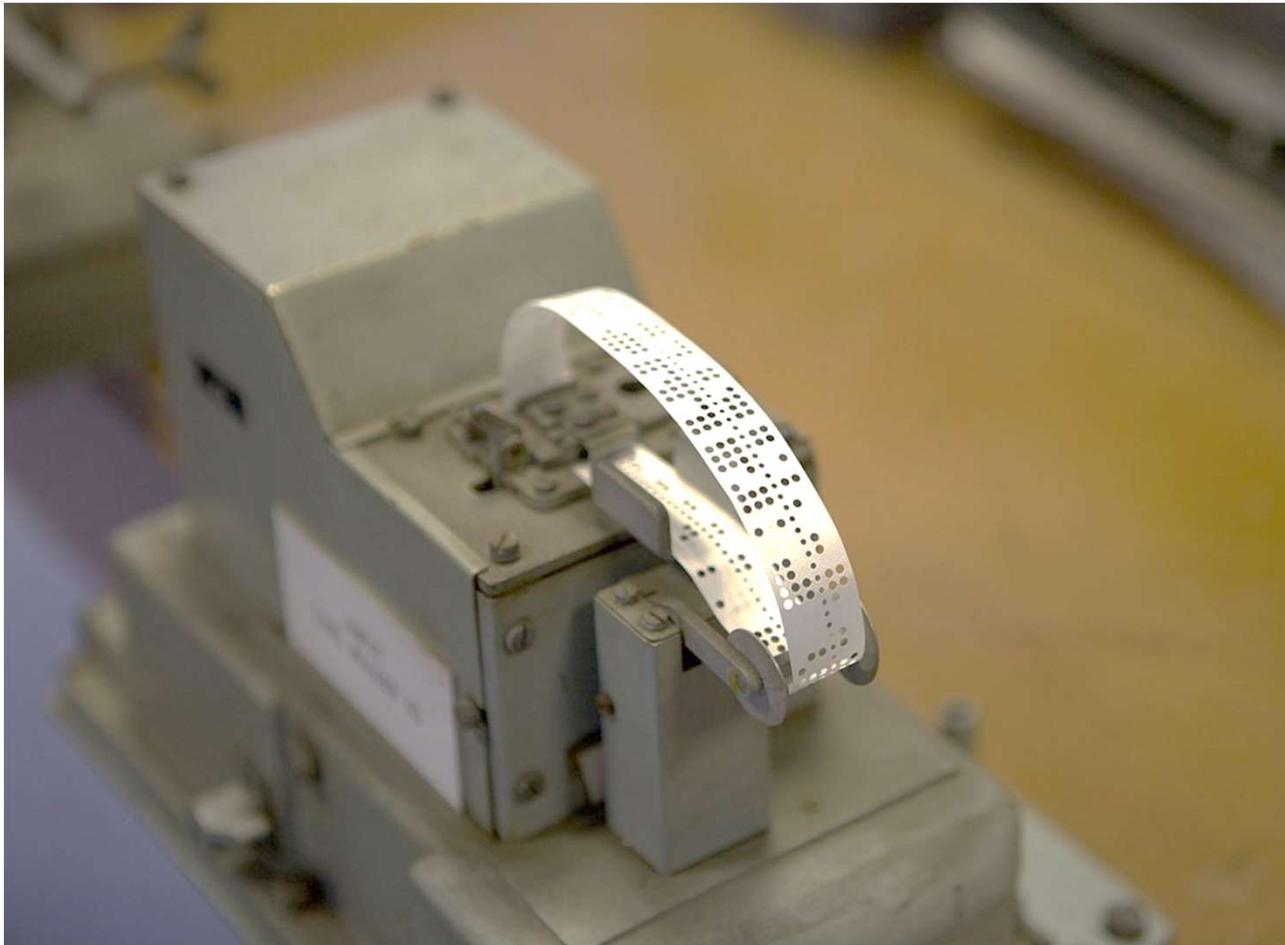
Let's try some basic things

- Function call
- Array access in a loop



What was the “pioneering” way?
What is the proper fix?

Technology Context Calibration



Evolution of Register Architecture

- Accumulator
 - carryover from adding machines and tabulators
 - no one wants more; no one can afford more
 - Accumulator + address registers
 - need register indirection (data and control-flow)
 - initially address registers were special-purpose,
i.e., only used to hold address for indirection
 - eventually arithmetic on address registers
 - General purpose registers (GPR)
 - all registers good for all purposes
 - grew from a few registers to 32 (common for RISC)
to 128 in Intel Itanium
- What drove the changes?

Operand Sources?

- Number of Specified Operands

Niladic	Op	(e.g. Burroughs)
Monadic	OP in2	(e.g. EDSAC)
Dyadic	OP inout, in2	(e.g. IBM 360)
Triadic	OP out, in1, in2	(e.g. MIPS)

- Can ALU operands be in memory?

No! e.g. MIPS/“RISC”/load-store arch.

Yes! e.g. x86/VAX/“CISC”

- How many different formats and addressing modes?

a very few e.g. MIPS / “RISC”

a lot e.g. x86

everything goes e.g. VAX

Memory Addressing Modes

- Absolute LW rd, 10000
use immediate value as address
- Register Indirect: LW rd, (r_{base})
use $\text{GPR}[r_{\text{base}}]$ as address
- Displaced or based: LW rd, offset(r_{base})
use $\text{offset} + \text{GPR}[r_{\text{base}}]$ as address
- Indexed: LW rd, ($r_{\text{base}}, r_{\text{index}}$)
use $\text{GPR}[r_{\text{base}}] + \text{GPR}[r_{\text{index}}]$ as address
- Memory Indirect LW rd ((r_{base}))
use value at $\text{M}[\text{GPR}[r_{\text{base}}]]$ as address
- Auto inc/decrement LW rd, ($r_{\text{base}}++$)
use $\text{GPR}[r_{\text{base}}]$ as address, and inc. or dec. $\text{GPR}[r_{\text{base}}]$
- Anything else you like to see

VAX-11: ISA in mid-life crisis

- First commercial 32-bit machine
- Ultimate in “orthogonality” and “completeness”

All of the above addressing modes x { 7 integer and 2 floating point formats} x {more than 300 opcodes}

- Opcode in excess
 - 2-operand and 3-operand versions of ALU ops
 - INS(/REM)QUE (for circular doubly-linked list)
 - “polyf”: 4th-degree polynomial solve
- Variable length encoding

addl3 r1,737(r2),(r3)[r4]

7-byte instruction, sequenced decoding

“RISC”

- Simple operations
 - 2-input, 1-output arithmetic and logical operations
 - few alternatives for accomplishing the same thing
 - Simple data movements
 - ALU ops are register-to-register (need large GPR file)
 - “load-store” architecture, 1 addressing mode
 - Simple branches
 - limited varieties of branch conditions and targets
 - Simple instruction encoding
 - all instructions encoded in the same number of bits
 - few, simple encoding formats
- motivated by/intended for compiled code over assembly

Evolution of ISAs

- Why were the earlier ISAs so simple? e.g., EDSAC
 - technology
 - precedence
- Why did it get so complicated later? e.g., VAX11
 - assembly programming
 - lack of memory size and speed
 - microprogrammed implementation
- Why did it become simple again? e.g., RISC
 - memory size and speed (cache!)
 - compilers
- Why is x86 still so popular?
 - technical merit vs. {SW base, psychology, deep pocket}
- Why has ARM thrived while other RISC ISAs vanished

CISC
Complex Instruction
Set Architecture

Reduced Instruction
Set Architecture

Why RISC-V now?

Major ISA Families

- 60s: **IBM 360**, DEC PDP-8
CDC 6000 (the original RISC)
- 70s: DEC PDP-11 → VAX
(CISCs) Intel **x86**, Motorola 680x0
6502, Z80, 8051
- 80s&90s: MIPS, **ARM**, SUN SPARC, HP PA-RISC,
(RISCs) IBM **Power**, Motorola 88K, DEC Alpha,
PowerPC (Apple+IBM+Motorola)
- 2000s: Intel IA-64
- 2010s: **RISC-V**

(overlooking embedded-only ISAs)

Intel IA-64/Itanium Architecture

- Late 90's attempt to counter RISC in servers market
- IA-64 Instruction “Bundle”
 - three IA-64 instructions (aka syllables)
 - template bits specify dependencies within a bundle and between bundles
 - group=collection of dependence-free bundles

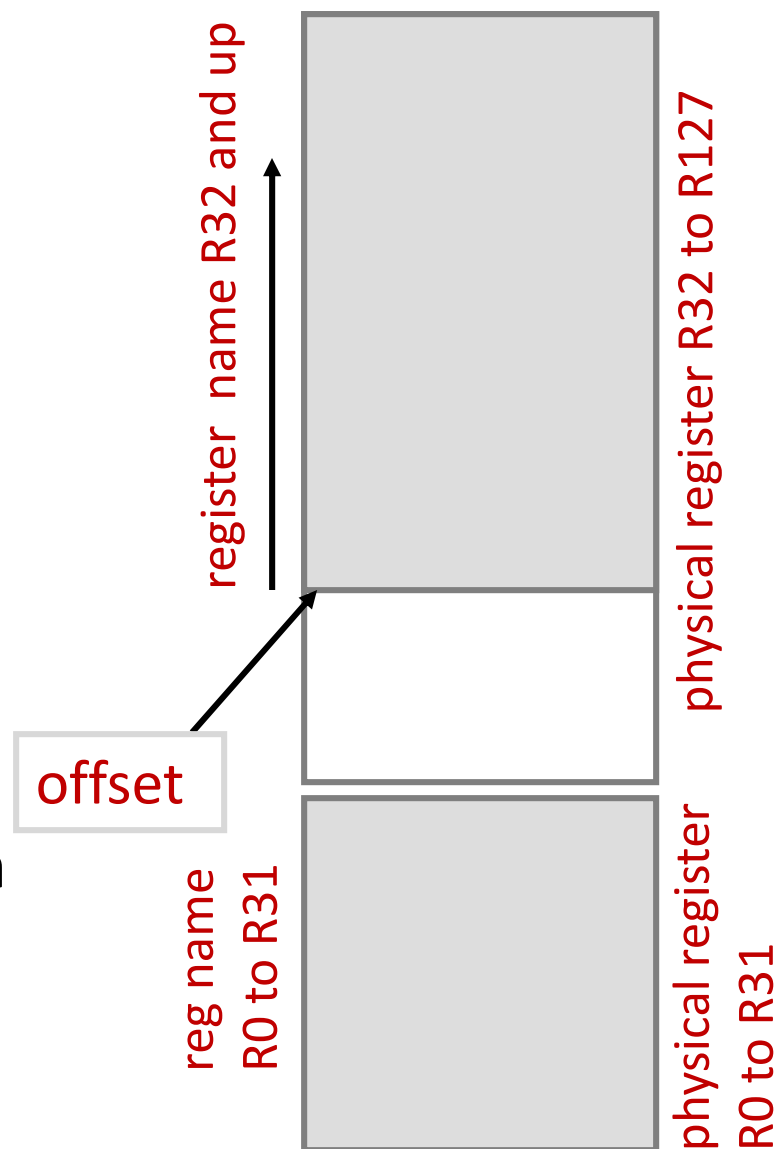
encode instruction parallelism explicitly



- “Thin” abstraction for simple/fast hardware
 - shift from dynamic HW to compiler static analysis and/or profile-driven
 - expose inst-by-inst performance mechanisms to SW
 - very hard to produce high performing code by hand

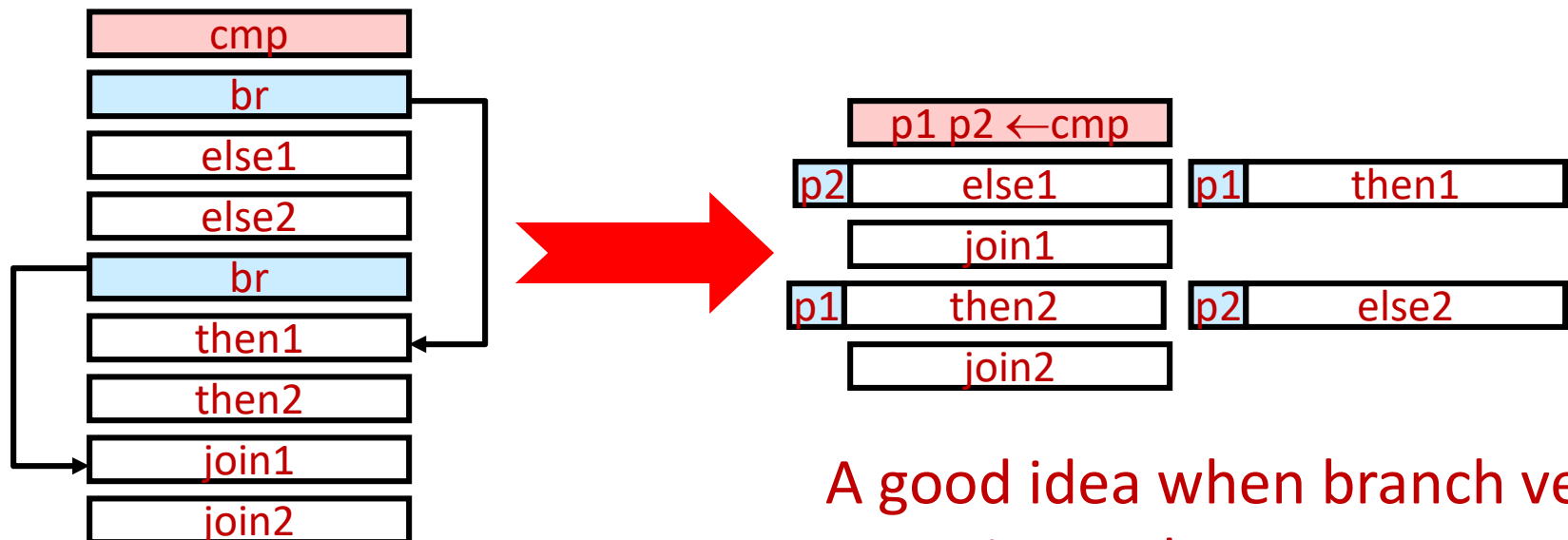
Example: Rotating Registers

- 128 general purpose physical integer registers
- Register names R0 to R31 are static; refer to the first 32 physical GPRs
- Register names R32 to R127 are “rotating registers”; renamed onto the remaining 96 physical registers by an offset
- Simplifies register use on function call/return and on loop optimizations (when register names are reused in code)



Example: Predicated Execution

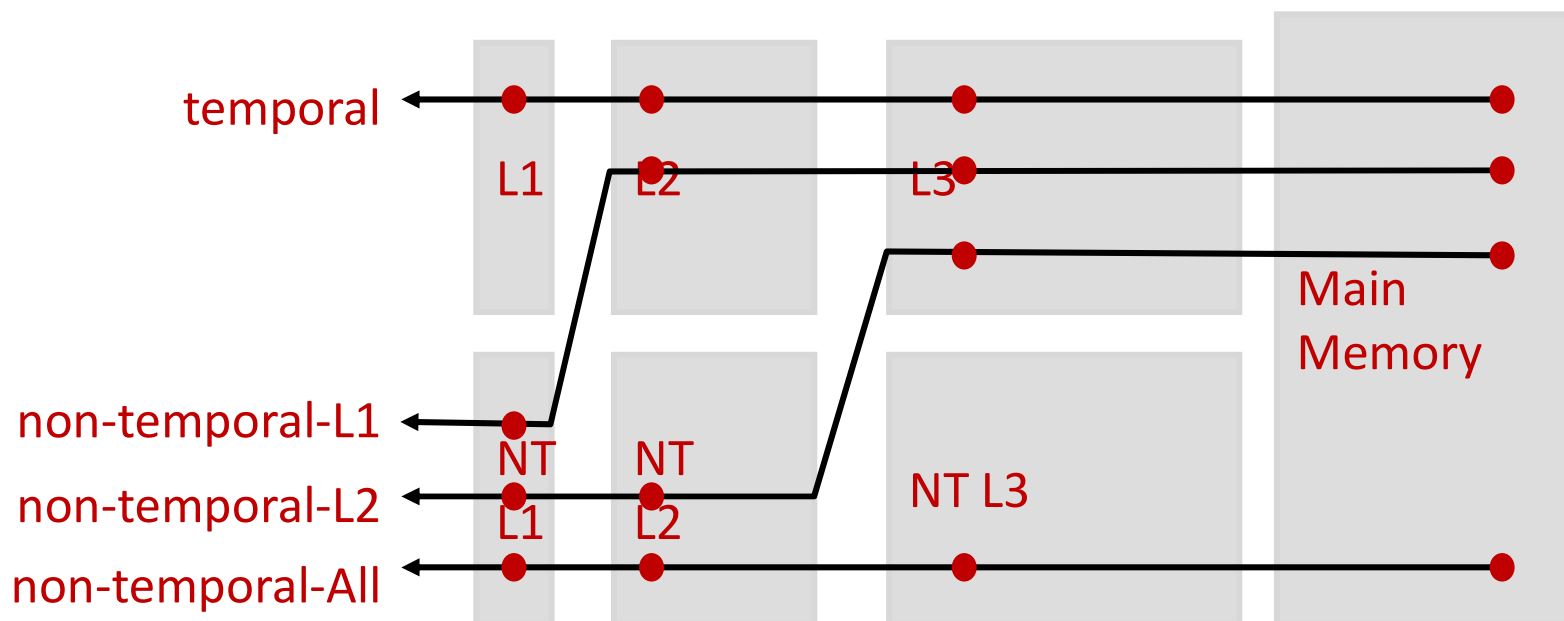
- 64 one-bit predicate register file
 - each instruction has a predicate register operand
 - instruction has no effect if predicate operand is false
- A way to realize conditionals without control flow



A good idea when branch very expensive and excess resources ready to absorb “extra” work

Example: Exposed Memory Hierarchies

- ISA included the concept of cache hierarchy
 - multiple levels
 - separate “temporal” vs “non-temporal”
 - Memory instructions give hints where best to cache
- As hints, microarchitecture does not have to comply



How much should ISA still matter?

Birth of “Binary Compatibility”

- “The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation.”

--- first defined in *Architecture of the IBM System/360*, Amdahl, Blaauw and Brooks, 1964.

- A single architecture with multiple price&perf variants replaced 4 incompatible product lines

Inter-Model Compatibility Defined

“a valid program whose logic will not depend implicitly upon time of execution and which runs upon configuration A, will also run on configuration B if the latter includes at least the required storage, at least the required I/O devices”

- Invalid programs not constrained to yield same result
 - “invalid” == violating architecture manual
 - “exceptions” are architecturally defined
- The King of Binary Compatibility: Intel x86, IBM 360
 - stable software base and ecosystem
 - performance scalability

[Amdahl, Blaauw and Brooks, 1964]

ISA Design Objective: General Purpose

- Effective support for “large and small, separate and mixed applications” in many domains
- Code-independent operation
 - no special interpretation of bit pattern in data
e.g. ASCII character has no special significance
 - except where essential
e.g., integer, floating point, etc.
- Support full generality of logic manipulation on bit and data entities
- Fine-grain memory addressability (down to small units of bits)

[Amdahl, Blaauw and Brooks, 1964]

ISA Design Objective: Open-Ended Design

“a dependable base for a decade of customer planning and customer programming . . .”

- Asynchronous operation of components—abstract out exact time, performance etc. to allow changing technology and relative speed of components
- Parameterization of storage capacity, multi CPU, multi I/O, etc.
- Standard interfaces for expansion sub-systems
- Permit future extensions by “reserving” spare bits in instruction encoding

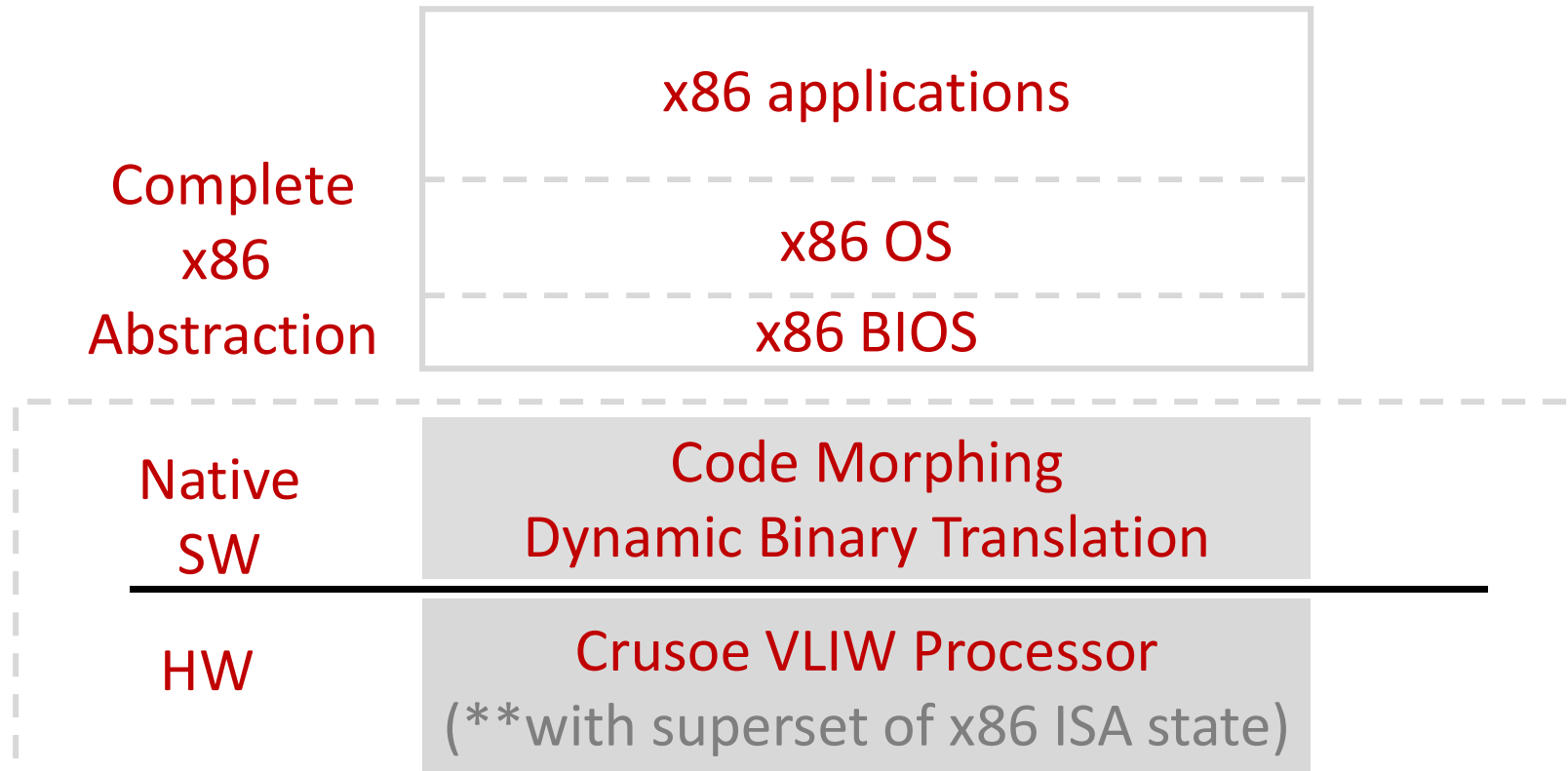
[Amdahl, Blaauw and Brooks, 1964]

What about Binary Translation

- Generate a new executable in **target ISA** with same functional behavior as the original in **source ISA**
 - not the same as interpretation or VM
 - not easy but doable (for the right source and target)
 - static vs dynamic
- Holy grail
 - all software run on the ISA/processor I sell
 - all processors can run the software I sell
- “Architecture” need not be the HW/SW contract
 - binary compatibility by translation virtualization
 - ISA and processor can become commodity
 - old software and ISA can live on for ever


What is CUDA/PTX?

Transmeta Crusoe & Code Morphing



- Crusoe boots “Code Morpher” from ROM at power-up
 - Crusoe+Code Morphing == x86 processor
- x86 software (including BIOS) cannot tell the difference
- BTW, this really worked in the early 2000s**

Code Morphing Software (CMS)

- Begins execution at power-up
 - fetches first-time x86 basic block from memory
 - translates BB into Crusoe VLIW and caches the translation for reuse
 - jumps to the generated Crusoe code for execution
 - continue directly from BB to BB if translation already cached; CMS regains control on new BBs
- 

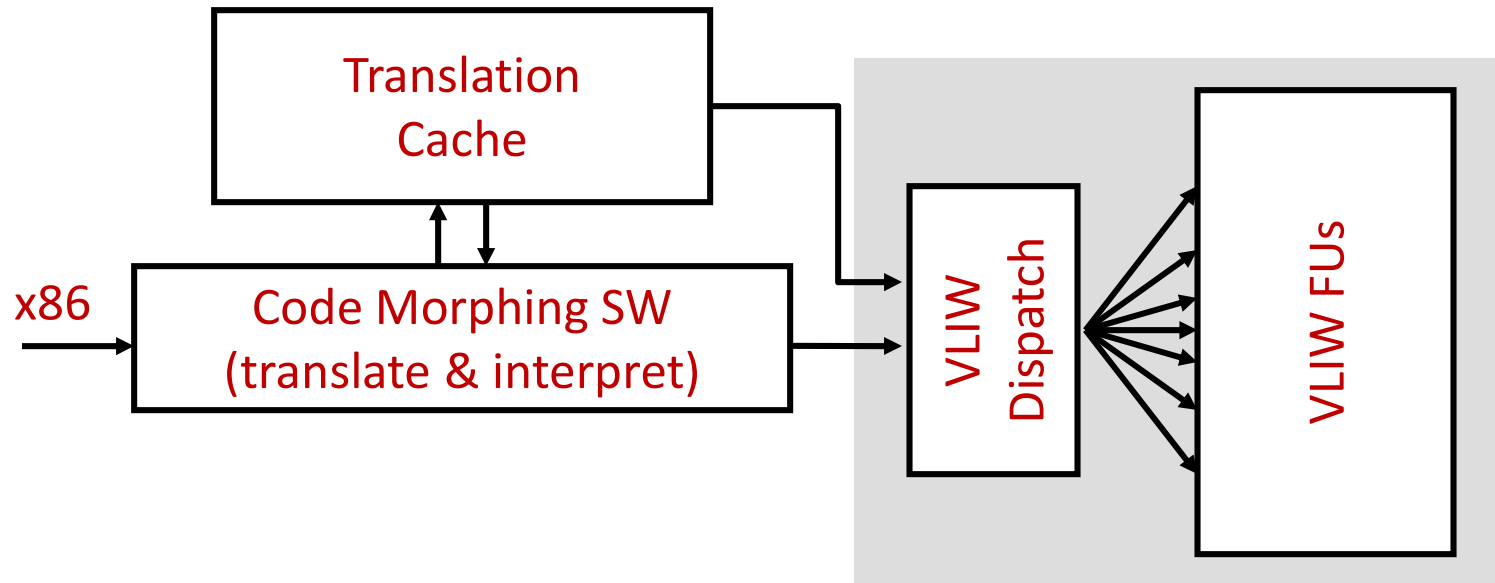
BB with “unsafe” x86 instructions not translated

- Re-optimize a translated block after runtime profiling
- The only native SW for Crusoe ISA

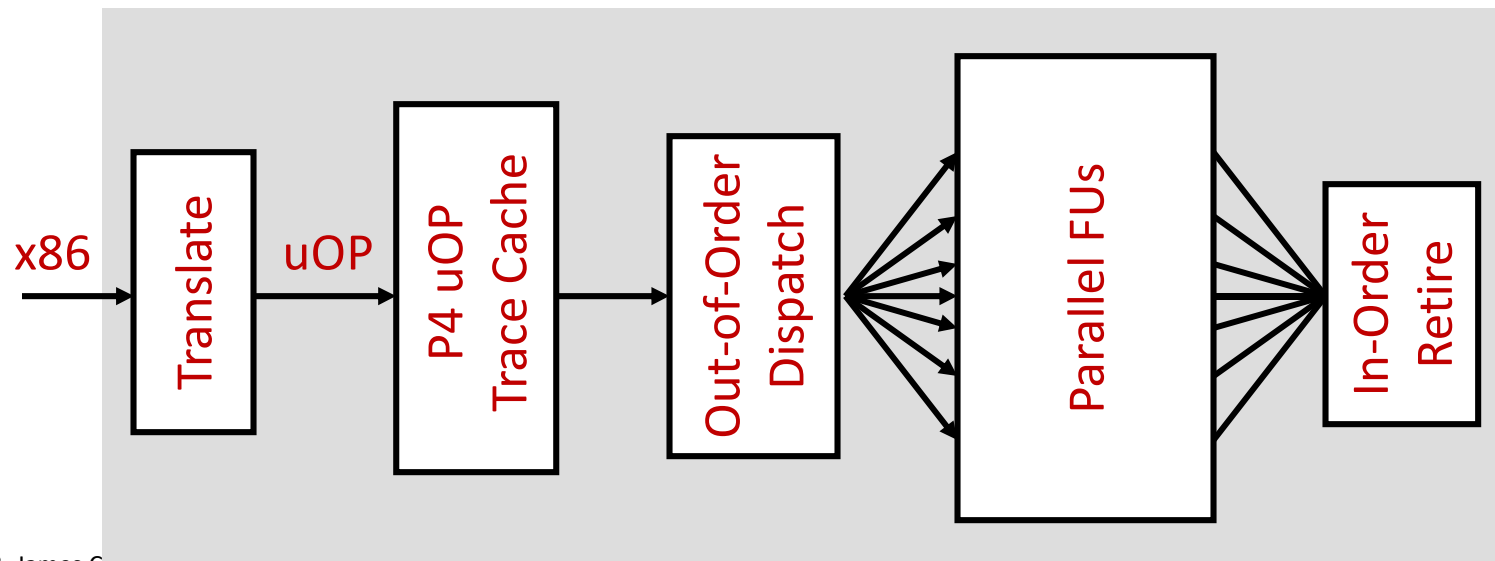
Crusoe processors do not need to be binary compatible between generations

Not really so different from Intel's own

Transmeta

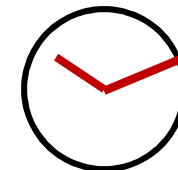


Intel Supercalar OOO



Stored Program Architecture

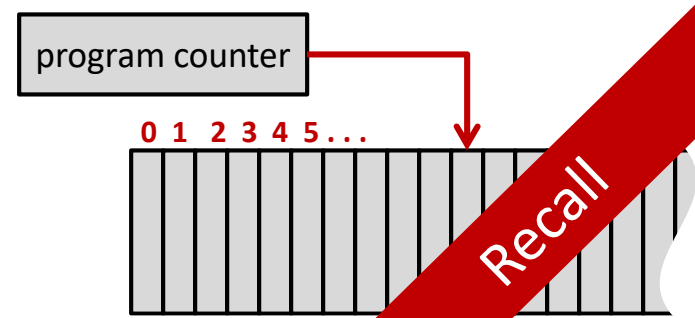
a.k.a. von Neumann



- Memory holds both program and data
 - instructions and data in a linear memory array
 - instructions can be modified as data
- Sequential instruction processing
 1. **program counter (PC)** identifies current instruction
 2. fetch instruction from memory
 3. update some state (e.g. **PC** and memory) as a function of current state (according to instruction)
 4. repeat



Dominant paradigm since its inception



von Neumann abstraction not free

- Significant transistor and energy overhead in presenting the simplifying abstraction
 - per-instruction access to program memory
 - dataflow through reading/writing of registers and memory state
 - “appearance” of sequentiality and atomicity
- In fact, von Neumann processors mostly overhead
- ISA future?
 - move away from von Neumann as doctrine?
 - do away with ISAs (lower-level, more explicit HW)?

Depend on what languages and compilers can do