Instruction Set Aspects

- format
  - length, encoding
- operations
  - operations, data types, number & kind of operands
- internal storage
  - model: accumulator, stack, general-purpose register
  - memory: address size, addressing modes, alignments
- control
  - branch conditions, special support for procedures, predication
- special features?

Instruction Format

fixed length (most common: 32-bits)

+ easy pipelining/superscalar
  - don't have to decode current instruction to find next instruction

– not compact (4-bytes for nop?)

variable length

+ more compact

– hard (but doable) to superscalarize/pipeline

recent compromise: 2 lengths (32-bit + another length)

- MIPS16, ARM Thumb: add 16-bit subset (compression)
- TM Crusoe: adds 64-bit long-immediate instructions

Operations

- arithmetic and logical: add, mult, and, xor
- data transfer: move, load, store
- control: conditional branch, jump, call, return
- system: system call, return, traps
- floating point: add, mul, div, sqrt
- decimal: addd, convert (not common today)
- string: move, compare (also not common)
- multimedia: e.g., Intel MMX/SSE and Sun VIS

Data Sizes and Types

- fixed point (integer)
  - 8-bit (byte), 16-bit (half), 32-bit (word), 64-bit (doubleword)
- floating point
  - 32/64 bit (IEEE754 single/double precision), 80-bit (Intel proprietary)
- address size (aka “machine size”)
  - e.g., 32-bit machine means addresses are 32-bits
  - key is virtual memory size: 32-bits $\Rightarrow$ 4GB (not enough anymore)

Fixed Point Operation Types

types: s/w (property of data) vs. h/w (property of operation)

- signed (-2^{n-1} to $2^{n-1}-1$) vs. unsigned (0 to 2^n-1)
- packed (multimedia short vector)
  - treat 64-bit as 8x8, 4x16, or 2x32
  - e.g.: addb, addh (MMX)

```
  17  87 100 ...
+ 17  13 200 ...
___ ___ ___ ___
 34 100  255 (saturating or 44 with wraparound)
```

  - MMX example: 16-element dot product: $\Sigma a_i * b_i$
  - plain: 200 instructions/76 cycles $\rightarrow$ MMX: 16/12 (6X perf.)
- saturating (no wrap around on overflow)
  - useful in RGBA calculations

Internal Storage Model

choices

- stack
- accumulator
- memory-memory
- register-memory
- register-register (load/store)

running example:

```
add C, A, B  (C := A + B)
```

Storage Model: Stack

```
push A   S[++TOS] = M[A];
push B   S[++TOS] = M[B];
add      T1=S[TOS--]; T2=S[TOS--]; S[++TOS]=T1+T2;
pop C    M[C] = S[TOS--];
```

- operands implicitly on top-of-stack (TOS, TOS2)
- ALU operations have zero explicit operands

\+ code density (top of stack implicit)

– memory, pipelining bottlenecks (why?)

- mostly 60's & 70's
  - x86 uses stack model for FP (sucks for them)
  - JAVA bytecodes also use stack model (why?)

Storage Model: Accumulator

```
load A    accum = M[A];
add B     accum += M[B];
store C   M[C] = accum;
```

- accum is implicit destination/source in all instructions
- ALU operations have one operand

+ less hardware, code density (accumulator implicit)

– memory bottleneck

- mostly pre 60's
  - UNIVAC, CRAY
  - x86 (IA32) uses extended accumulator for integer code
- accumulator comeback?
  - 2-level register file (register-accumulator) [ISCA'02]

Storage Model: Memory-Memory

```
add C,A,B   M[C] = M[A] + M[B];
```

- no registers whatsoever

+ code density (most compact)

– large variations in instruction lengths

– large variations in work per-instruction

– memory bottleneck

- no current machines support memory-memory
  - VAX did

Storage Model: Memory-Register

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

- like an explicit (extended) accumulator
  - + can have several accumulators at a time
- + code density, easy to decode
- – asymmetric operands, asymmetric work per instruction
- 70’s and early 80’s
  - IBM 360/370
  - Intel x86, Motorola 68K

Storage-Model: Register-Register

```
load R1,A      R1 = M[A];
load R2,B      R2 = M[B];
add R3,R1,R2   R3 = R1 + R2;
store C,R3     M[C] = R3;
```

- load/store architecture: ALU operations on registers only

– code density

+ easy decoding, operand symmetry

+ deterministic length ALU operations

+ scheduling opportunities, register-level CSE

- 60’s and onwards
  - RISC machines: ALPHA, MIPS, PPC (but also Cray)

Registers vs. Memory

registers

- \+ faster (direct access, smaller, no tags)
- \+ deterministic scheduling (i.e., fixed latency, no misses)
- \+ replicate for more b/w
- \+ short identifier
- – must save/restore on procedure calls, context switches
- – fixed size
  - strings, structures (i.e., bigger than 64 bits) *must* live in memory
- – can’t take address of a register
  - pointed-to variables *must* live in memory

How Many Registers?

more registers

+ hold more operands for longer periods
  - shorter average operand access time, lower memory traffic

– longer specifiers (longer instructions?)

– slower access to register operands (bigger is slower)

– slower procedure calls/context-switch (more save/restore)

trend is for *more* registers

- X86: 8 → SPARC/MIPS/Alpha/PPC: 32 → Itanium: 128
  - why?

Memory Alignment

"natural boundaries" ⇒ (address % size) == 0

- e.g. word (4 bytes): @xx00 ⇒ aligned, @xx11 ⇒ unaligned

alignment restrictions: kinds of alignments architecture supports

- no restrictions (all in hardware)
  - hardware detects, makes 2 references (what if 2nd one faults?)
  - – expensive logic, slows down all references (why?)
- restricted alignment (software guarantee w/ hardware trap)
  - misaligned access traps, performed in s/w by handler
- middle ground: multiple instructions for misaligned data
  - e.g., MIPS (lwl/lwr), Alpha (ldq_u)
  - compiler generates for known cases, h/w traps for unknown cases

Operand Addressing Modes

- immediate: #n (immediate values)
- register: R_i (register values)
- displacement: M[R_i + #n] (stack, structure fields)
- register indirect: M[R_i] (loaded/computed addresses)
- memory absolute: M[#n] (globals)
- indexed: M[R_i + R_j] (arrays of scalars)
- memory indirect: M[M[R_i]] (in-memory pointers)
- scaled: M[R_i + R_j * d + #n] (arrays of structures, X86)
- update/auto-increment/decrement: M[R_i = R_i + #n]

Operand Addressing Modes

1–4 account for 93% of all VAX operands [Clark+Emer]

RISC machines typically implement 1–3

- i.e., load/store with only register displacement
  - load: $R_j = M[R_i + \#n]$, store: $M[R_i + \#n] = R_j$
- synthesize all other modes
  - e.g., memory indirect: $R_j = M[M[R_i]] \Rightarrow R_k = M[R_i]; R_j = M[R_k]$

Control Instructions

aspects

- 1. taken or not?
- 2. where is the target?
- 3. link return address?
- 4. save or restore state?

instructions that change the PC

- (conditional) branches [1, 2], (unconditional) jumps [2]
- function calls [2,3,4], function returns [2,4]
- system calls [2,3,4], system returns [2,4]

Taken or Not?

- “compare and branch” instructions
  + single instruction branches
  – requires ALU op in branch pipeline, restricts scheduling
- separate “compare” and “branch” instructions
  – uses up a register, separates condition from branch logic
  + more scheduling opportunities, reuse comparison
- condition codes: Zero, Negative, oVerflow, Carry
  + set “for free” by ALU operations
  – extra state to save/restore, scheduling problems (implicit)
- MIPS example (design instruction set for usage)
  - data: 80+% cmp immediate, 65+% cmp zero, 50% == 0 or <> 0
  - ISA: beqz, bnez, compare and set + branch for others

Where is the Target?

- PC-relative: branches/jumps within function
  - + position independent, computable early, #bits: <4 (47%), <8 (94%)
  - – target must be known statically, can’t jump far
- absolute: function calls, long jumps within functions
  - + jump farther
  - – more bits to specify
- register: indirect calls (DLLs, virtuals), returns, switch
  - + short specifier, can jump anywhere, dynamic target ok (ret)
  - – extra instruction (load), branch and target separated in pipeline
- vectored trap: system calls
  - + protection
  - – surprises are implementation headache

Link Return Address?

- implicit register: many recent architectures use this
  - + fast, simple
  - – s/w save register before next call (pain: surprise trap)
- explicit register
  - + may avoid saving register
  - – register must be specified
- processor stack
  - + recursion direct
  - – complex instructions (yucky)

Save or Restore State?

- function calls: save/restore registers
- system calls: save/restore registers, flags, PC, PSW, etc.
- software save/restore: calling convention divides work
  - *caller* saves registers in use
  - *callee* saves registers it (or nested callees) will use
- explicit hardware save/restore
  - IBM STM, VAX CALLS
- implicit hardware save/restore: register windows (SPARC)
  - 32 registers: 8 in, 8 out, 8 local, 8 global
  - call: out in (pass parameter), local/out “fresh”, global unchanged
  - on return: opposite, 8 output of caller restored
  - saving/restoring to memory when h/w windows run out

  + no saving/restoring for shallow call graphs

  – make register renaming (needed for OoO execution) hard

Alternative to Control: Predication

if (a > 0) c = b*a;

```
#0: blez r1, #2
#1: mul r3, r2, r1
```

- problem? #0 is a branch
  - – expensive if mis-predicted (later)
- predication: converts control-flow to data-flow
  - + branch mis-prediction avoided
  - – but data-dependences complicated
  - • two ways: conditional moves (left), or general predication (right)

```
#0: mul r4, r2, r1
#1: cmovgt r3, r4, r1
```

```
#0: sgtzp p1, r1
#1: divp r3, r4, r1, p1
```

RISC War: RISC vs. CISC

early 80's: RISC movement challenges "CISC establishment"

- RISC (reduced instruction set computer)
  - Berkeley RISC-I (Patterson), Stanford MIPS (Hennessy), IBM 801
- CISC (complex instruction set computer)
  - VAX, X86, etc.
- word CISC did not exist before word RISC came along

RISC Manifesto

- single-cycle operation (CISC: many multi-cycle ops)
- hardwired control (CISC: microcode)
- load/store organization (CISC: mem-reg, mem-mem)
- fixed instruction format (CISC: variable format)
- few modes (CISC: many modes)
- reliance on compiler optimization (CISC: hand assembly)
  - + load/store ⇒ register allocation (+21% performance)
  - + simple instructions ⇒ fine-grain CSE (+10%), scheduling (?)

- no equivalent “CISC manifesto”

The Joke on RISC

most commercially successful ISA is x86 (decidedly CISC)

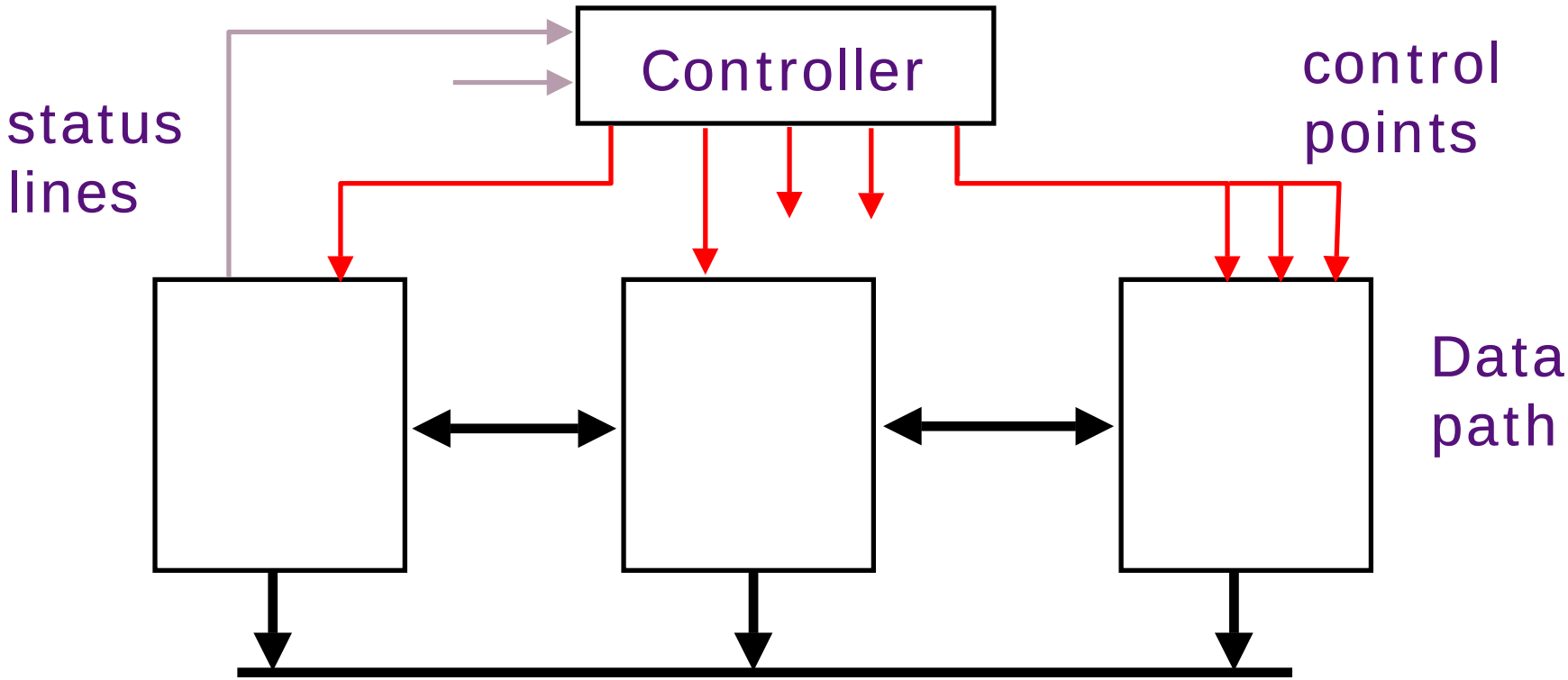
- also: PentiumPro was first out-of-order microprocessor
  - good RISC pipeline, 100K transistors
  - good CISC pipeline, 300K transistors
  - by 1995: 2M+ transistors evened pipeline playing field
  - rest of transistors used for caches (diminishing returns)
- Intel's other trick?
  - decoder translates CISC into sequences of RISC μops

Microprogramming

ISA to Microarchitecture Mapping

- An ISA often designed for a particular microarchitectural style, e.g.,
  - CISC ⇒ microcoded
  - RISC ⇒ hardwired, pipelined
  - VLIW ⇒ fixed latency in-order pipelines
  - JVM ⇒ software interpretation
- But an ISA can be implemented in any microarchitectural style
  - Pentium-4: hardwired pipelined CISC (x86) machine (with some microcode support)
  - This lecture: a microcoded RISC (MIPS) machine
  - Intel will probably eventually have a dynamically scheduled out-of-order VLIW (IA-64) processor
  - PicoJava: A hardware JVM processor

Microarchitecture: *Implementation of an ISA*



Structure: How components are connected.

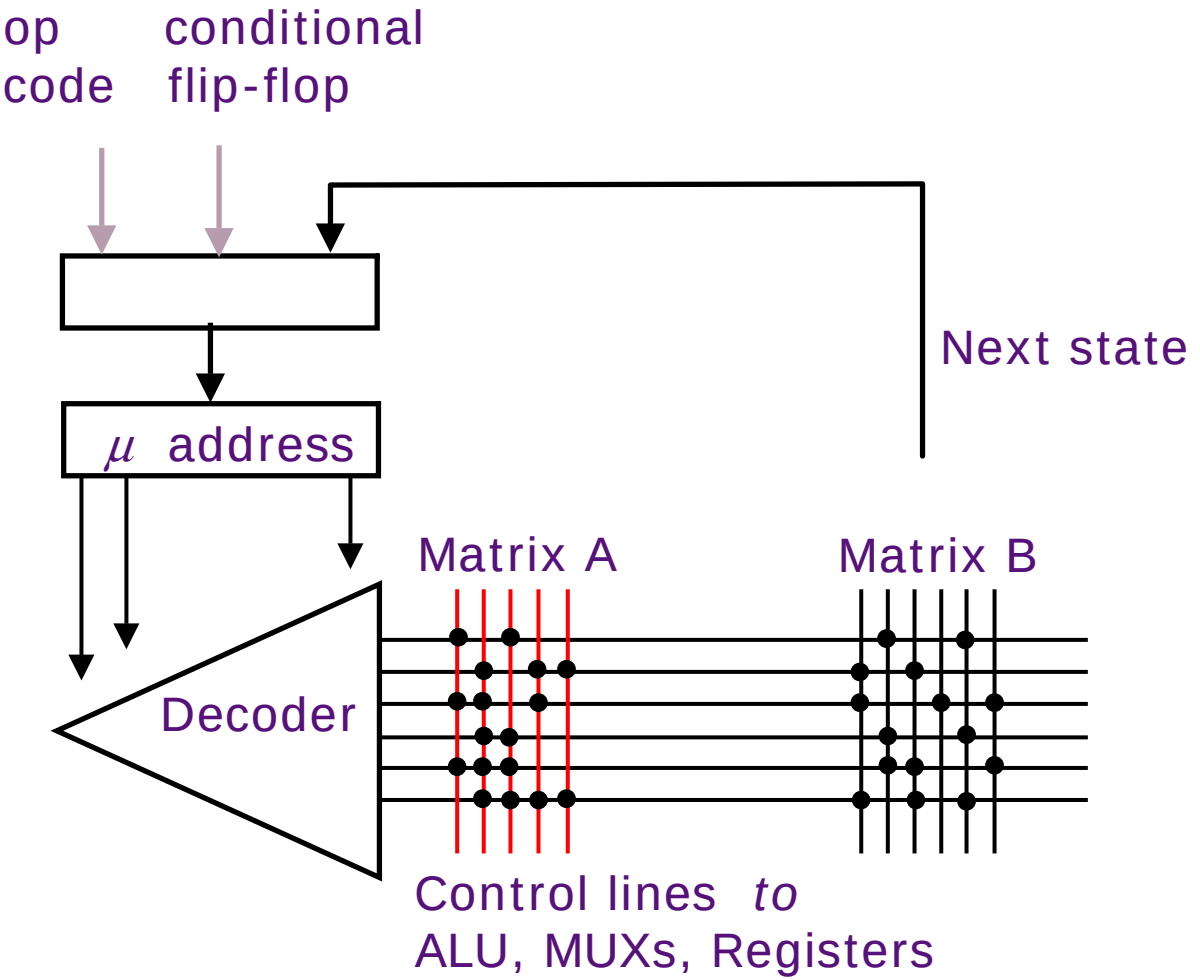
Static

Behavior: How data moves between components

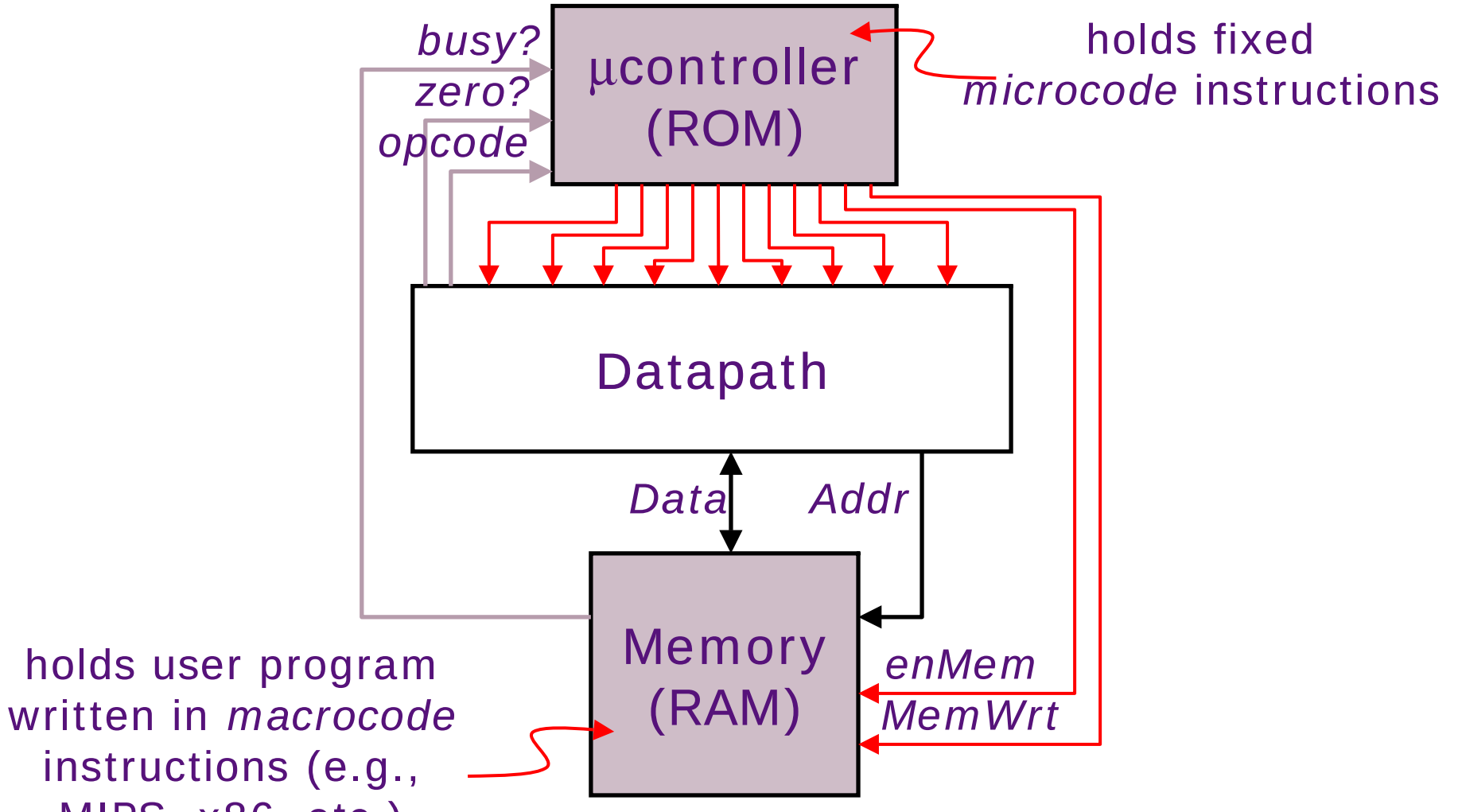
Dynamic

Microcontrol Unit *Maurice Wilkes, 1954*

Embed the control logic state table in a memory array



Microcoded Microarchitecture



Reducing Control Store Size

Control store has to be *fast* $\Rightarrow$ *expensive*

- Reduce the ROM height (= address bits)
  - *reduce inputs by extra external logic*
    each input bit doubles the size of the control store
  - *reduce states by grouping opcodes*
    find common sequences of actions
  - *condense input status bits*
    combine all exceptions into one, i.e., exception/no-exception
- Reduce the ROM width
  - *restrict the next-state encoding*
    Next, Dispatch on opcode, Wait for memory, ...
  - *encode control signals (vertical microcode)*

Performance Issues

Microprogrammed control

$\Rightarrow$ multiple cycles per instruction

Cycle time ?

$t_C > \max(t_{reg\text{-}reg}, t_{ALU}, t_{\mu ROM}, t_{RAM})$

Given complex control, t_{ALU} & t_{RAM} can be broken into multiple cycles. However, $t_{\mu ROM}$ cannot be broken down. Hence

$t_C > \max(t_{reg\text{-}reg}, t_{\mu ROM})$

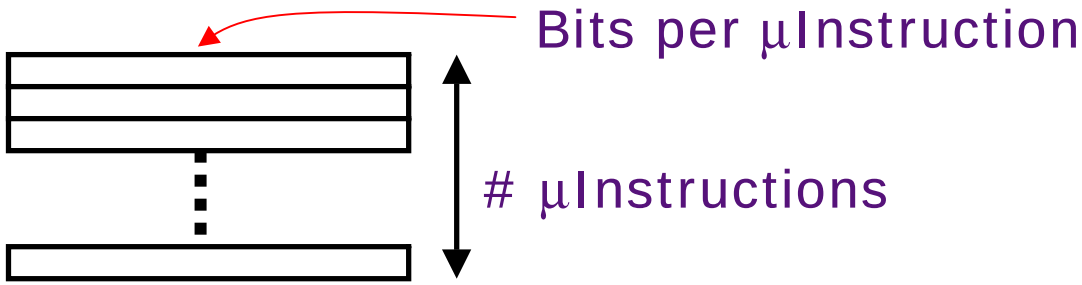
Suppose $10 * t_{\mu ROM} < t_{RAM}$

Good performance, relative to the single-cycle hardwired implementation, can be achieved even with a CPI of 10

Modern Usage

- *Microprogramming is far from extinct*
- Played a crucial role in micros of the Eighties
  DEC uVAX, Motorola 68K series, Intel 386 and 486
- Microcode pays an assisting role in most modern CISC micros *(AMD Athlon, Intel Pentium-4 ...)*
  - Most instructions are executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke the microcode engine
- *Patchable* microcode common for post-fabrication bug fixes, e.g. Intel Pentiums load mcode patches at bootup

Horizontal vs Vertical μCode



- Horizontal μcode has wider μinstructions
  - Multiple parallel operations per μinstruction
  - Fewer steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical μcode has narrower μinstructions
  - Typically a single datapath operation per μinstruction
    - separate μinstruction for branches
  - More steps to per macroinstruction
  - More compact ⇒ less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical μcode

Evolution of Instruction Sets

instruction sets evolve

- implementability driven by technology
  - microcode, VLSI, pipelining, superscalar
- programmability driven by (compiler) technology
  - hand assembly → compilers → register allocation
- instruction set features go from good to bad to good
  - just like microarchitecture ideas

lessons

- many non-technical (tr: business) issues influence ISAs
- best solutions don’t always win