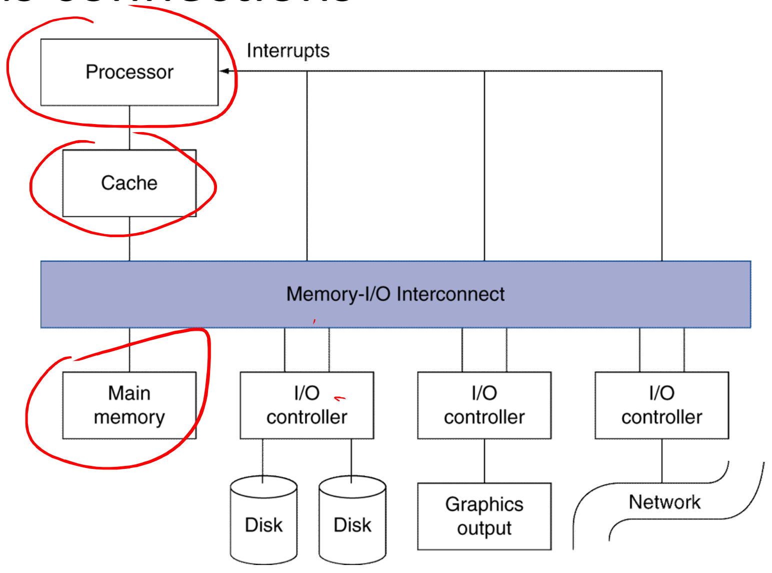
Input and Output

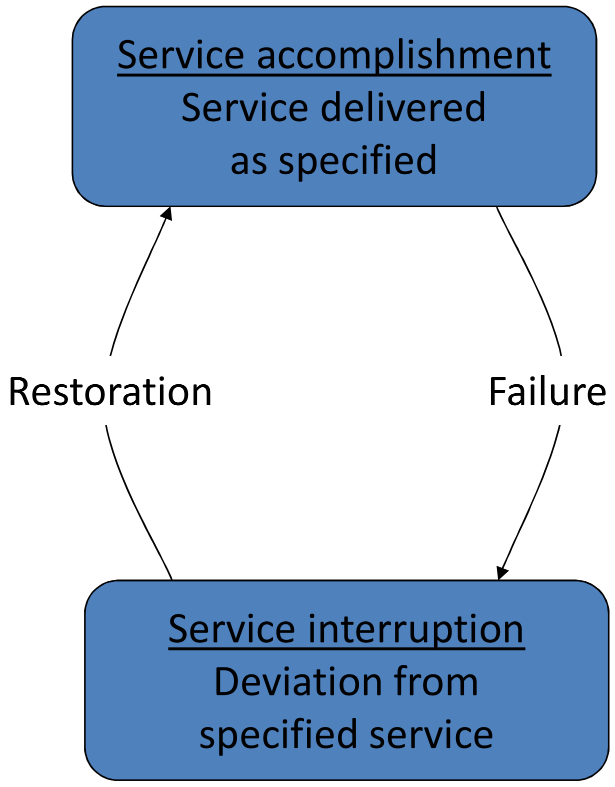
- I/O devices can be characterized by
  - Behavior: input, output, storage
  - Partner: human or machine
  - Data rate: bytes/sec, transfers/sec
- I/O bus connections



I/O System Characteristics

- Dependability is important
  - Particularly for storage devices
- Performance measures
  - Latency (response time)
  - Throughput (bandwidth)
  - Desktops & embedded systems
    - Mainly interested in response time & diversity of devices
  - Servers
    - Mainly interested in throughput & expandability of devices

Dependability


Service accomplishment
Service delivered as specified
Restoration
Failure
Service interruption
Deviation from specified service

- Fault: failure of a component
  - May or may not lead to system failure

Dependability Measures

- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

I/O

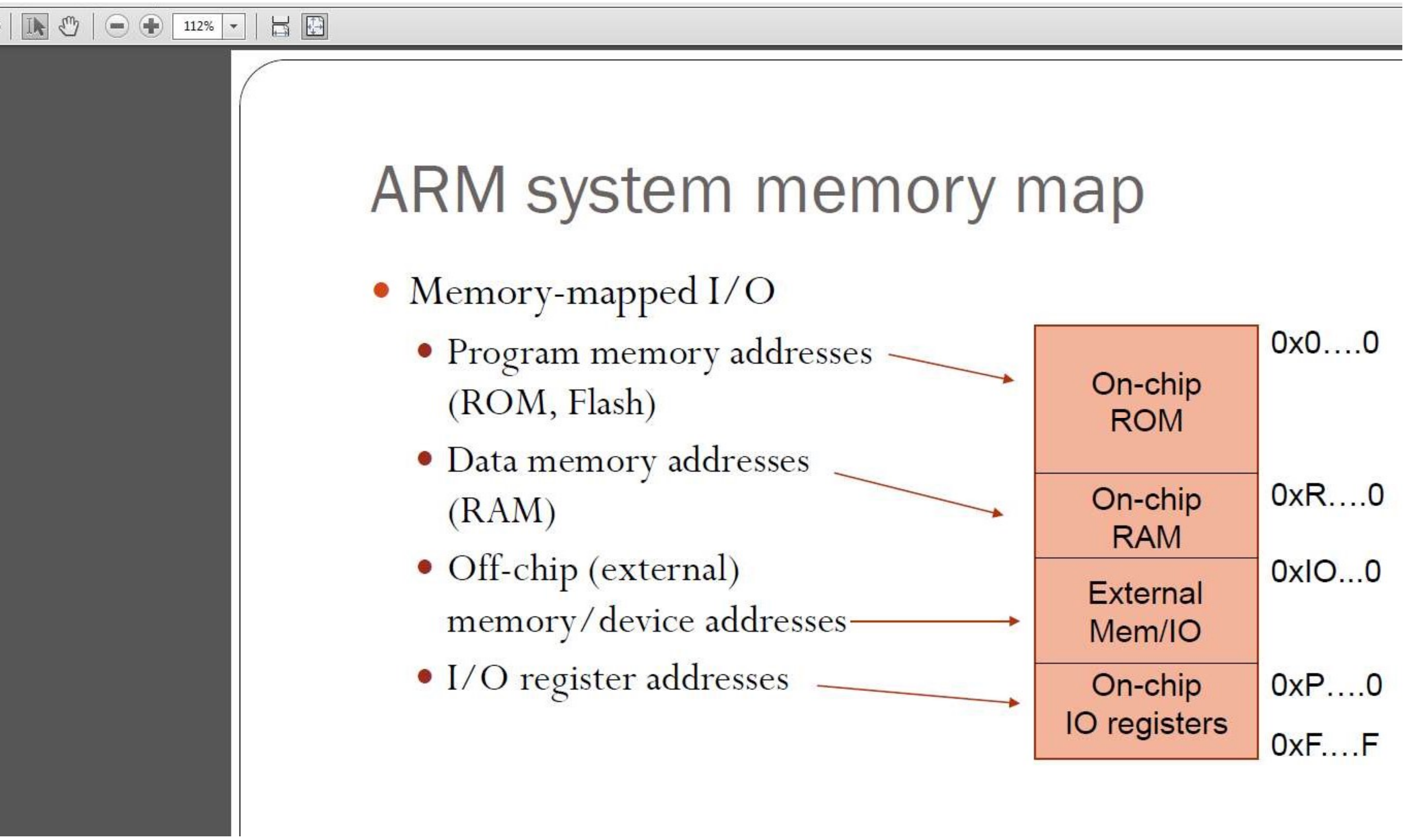
- Indispensable part of the computer systems
    - Dependability vs performance
    - More diverse (variety of IO devices)
- Memory-mapped I/O

Programming I/O

Two types of instructions can support I/O:
- special-purpose I/O instructions
- memory-mapped load/store instructions

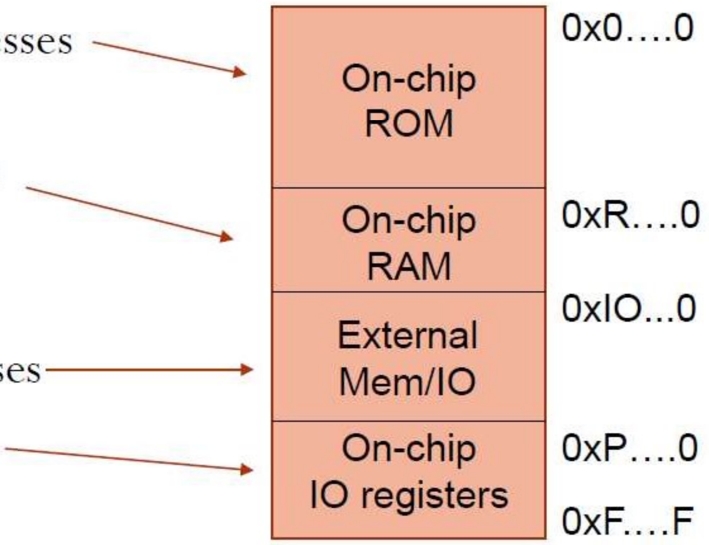
Intel x86 uses in, out instructions

Most CPUs use memory-mapped I/O ✓



ARM system memory map

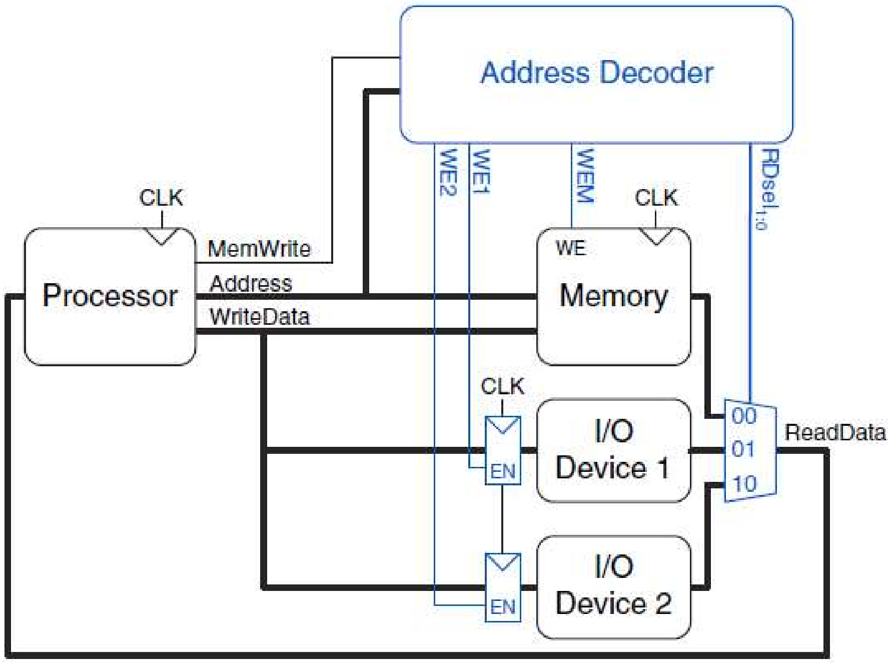
- Memory-mapped I/O
  - Program memory addresses (ROM, Flash)
  - Data memory addresses (RAM)
  - Off-chip (external) memory/device addresses
  - I/O register addresses



Memory-mapped I/O

e.g., suppose I/O device1 is assigned mem. Addr. 0xFFFFFFF4. Then write a value 7 to the device is accomplished by the following instructions.

```
LDR     r1, =0xFFFFFFF4
MOV    r0, #7
STRB   r0, [r1]
```



Programmed I/O

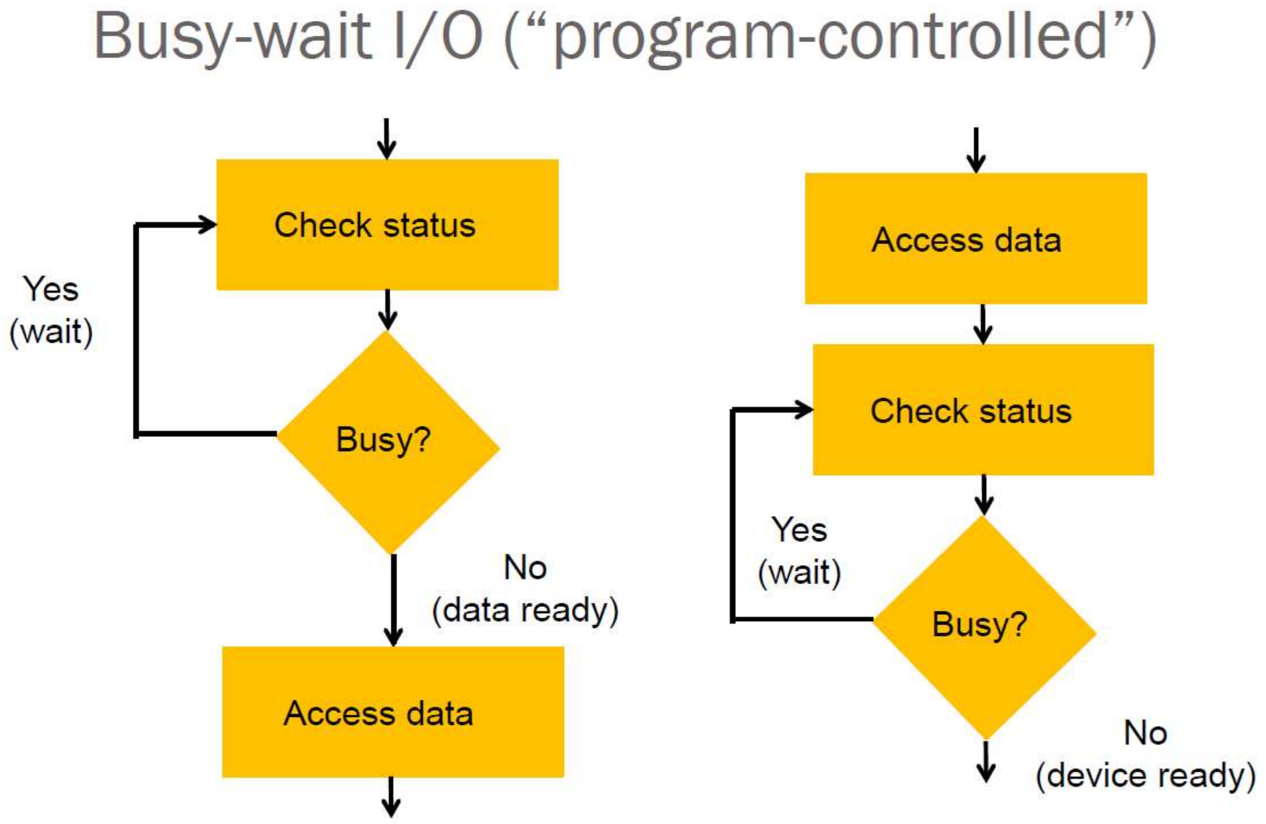
IA-32 uses specialized instructions instead of memory mapped

e.g.,

```
lwio  $t0, device1
swio  $t0, device2
```

Where device1 and device2 are unique id for I/O devices.

Pooling


Busy-wait I/O ("program-controlled")
Check status
Yes
(wait)
Busy?
No
(data ready)
Access data
Access data
Check status
Yes
(wait)
Busy?
No
(device ready)

```
/* send a character string */
current_char = mystring;    //char string ptr
while (*current_char != '\0') {
  OUT_CHAR = *current_char; //write a character
  while (OUT_STATUS != 0);  //wait while busy
  current_char++;
}
```

```
        LDR     r1, =0x40000100     @address for device status
W:      LDRB    r2, [r1]
        TST     r2, #1
        BEQ     W
        LDR     r3, =0x40000104     @address for output device
        STRB    r0, [r3]
```

Simultaneous busy/wait input and output

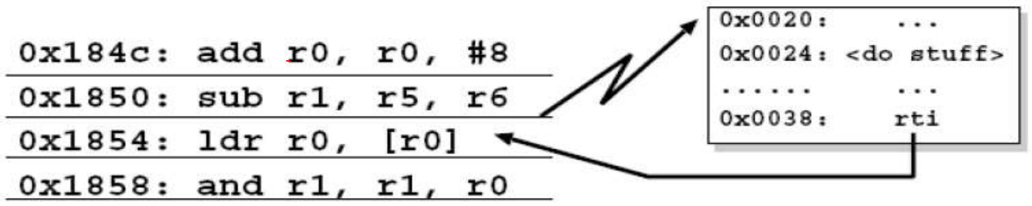
```
while (TRUE) {
  /* read */
  while (IN_STATUS == 0);  //wait until ready
  achar = IN_DATA;         //read data
  /* write */
  OUT_DATA = achar;        //write data
  while (OUT_STATUS != 0); //wait until ready
  }
```

Pooling Busy/Wait is inefficient

- CPU cannot do other work while testing device readiness
- Hard to carry out simultaneous I/O

Interrupts

- Bus reads and writes are *transaction* based: CPU requests something and waits until it happens.
- But e.g. reading a block of data from a hard-disk takes ~2ms, which might be over 10,000,000 clock cycles!
- ***Interrupts*** provide a way to decouple CPU requests from device responses.
  1. CPU uses bus to make a request (e.g. writes some special values to addresses decoded by some device).
  2. Device goes off to get info.
  3. Meanwhile CPU continues doing other stuff.
  4. When device finally has information, raises an ***interrupt***.
  5. CPU uses bus to read info from device.
- When interrupt occurs, CPU ***vectors*** to handler, then ***resumes*** using special instruction, e.g.



```
0x184c: add r0, r0, #8
0x1850: sub r1, r5, r6
0x1854: ldr r0, [r0]
0x1858: and r1, r1, r0
```

```
0x0020:     ...
0x0024: <do stuff>
......      ...
0x0038:     rti
```

Other Processors and ISA’s

- MIPS
- Intel (IA-32)

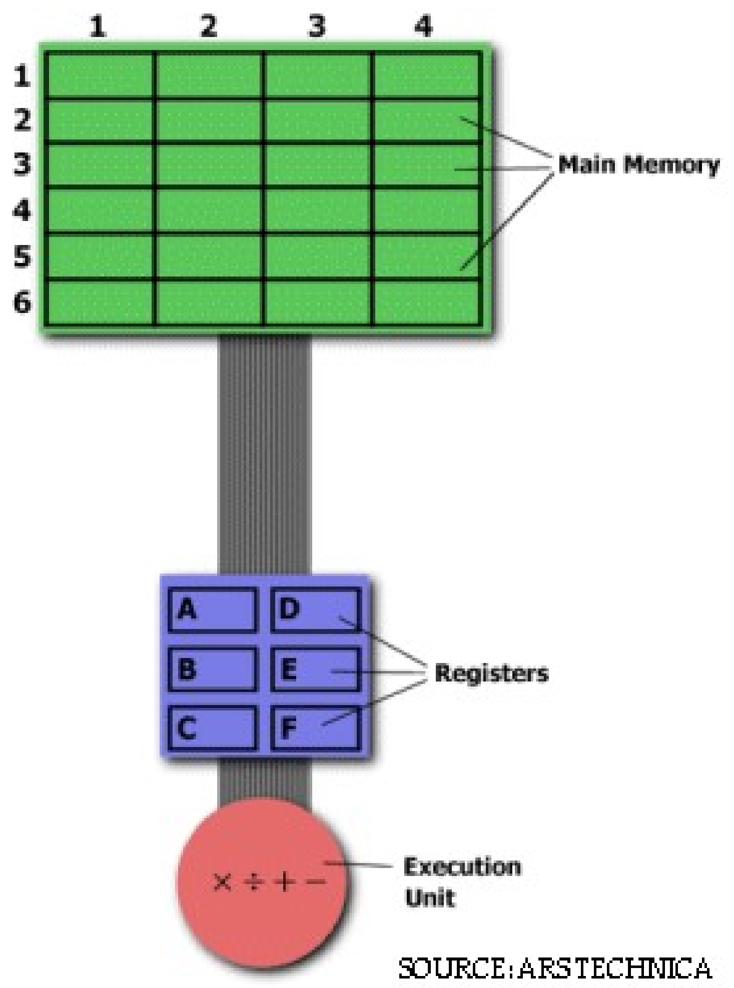
ISA

Well defined hardware/software interface

- The "contract" between software and hardware
  - Functional definition of storage locations and operations
  - Precise description of how to invoke and access
- Not in contract: non-functional aspects
  - How operations are implemented
  - Which operations are fast

Good ISA:

- Programmability
  - Easy to express programs efficiently
- Performance and Implementability
  - Easy to design: high-performance, low energy consumption and low-cost
- Compatibility
  - Easy to maintain as languages, programs and technology evolve
  - X86 (IA32) generations: 8086, 286, 386, 486, Pentium, …, core i7



C code:

a = a * b

Assembly code:
CISC ISA

```
MULT   2:3, 5:2
```

RISC ISA

```
LDR    A, 2:3
LDR    B, 5:2
MULT   C, A, B
STR    C, 2:3
```

Instruction Length and Encoding

- **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 averages 3 bytes, ranges from 1 to 16)
    - – 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 nasty piece of logic

- **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 started in mid 1980's
  - RISC "won" the technology battles
  - CISC won the high-end commercial space (1990s to today)
    - Compatibility was a strong force
  - RISC winning the embedded computing space

HOME SEARCH The New York Times SUBSCRIBE NOW LOG IN

Computer Chip Visionaries Win Turing Award | TECH TIP Sharing Your Story in Instagram | Toyota Takes Self-Driving Cars Off Road After Uber Accident | The End for Facebook's Security Evangelist | TRENDING 1. John Dowd Resigns as Trump's Lead Lawyer in Special Counsel Inquiry | 2. Why He Kayaked Across the Atlantic at 70 (for the Third Time)

TECHNOLOGY

Computer Chip Visionaries Win Turing Award

By CADE METZ MARCH 21, 2018

Dave Patterson, right, and John Hennessy in the early 1990s. The men won the Turing Award for their pioneering work on a computer chip design that is now used by most of the tech industry.
Shane Harvey

SAN FRANCISCO — In 1980, Dave Patterson, a computer science professor, looked at the future of the world's digital machines and saw their limits.

With an academic paper published that October, he argued that the silicon chips at the heart of these machines were growing more complex with each passing year. But the machines, he argued, could become more powerful if they used a simpler type of computer chip.

RELATED COVERAGE

Chips Off the Old Block: Computers Are Taking Design Cues From Human Brains
SEPT. 16, 2017

Big Bets on A.I. Open a New Frontier for

- Performance equation: **CPU Time = IC x CPI x CC**
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)

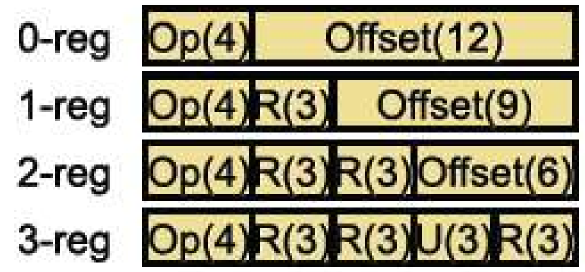
- **CISC** (Complex Instruction Set Computing)
  - Reduce "instructions/program" with "complex" instructions
    - But tends to increase "cycles/instruction" or clock period
  - Easy for assembly-level programmers, good code density
- **RISC** (Reduced Instruction Set Computing)
  - Improve "cycles/instruction" 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**

RISC Design Principles

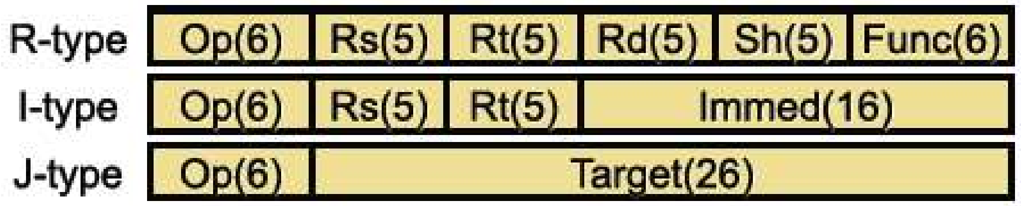
- **Single-cycle execution**
  - CISC: many multicycle operations
- **Hardwired (simple) control**
  - CISC: "microcode" for multi-cycle operations
- **Load/store architecture**
  - CISC: register-memory and memory-memory
- **Few memory addressing modes**
  - CISC: many modes
- **Fixed-length instruction format**
  - CISC: many formats and lengths
- **Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance
- **Many registers** (compilers can use them effectively)
  - CISC: few registers

CISC	**RISC**
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

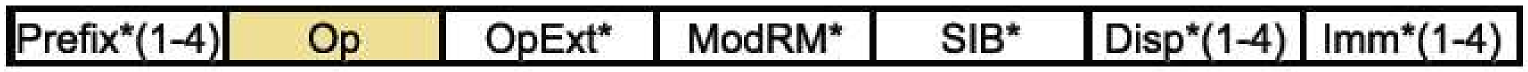
- LC4: 2-byte insns, 3 formats



- MIPS: 4-byte insns, 3 formats



- x86: 1–16 byte insns, many formats

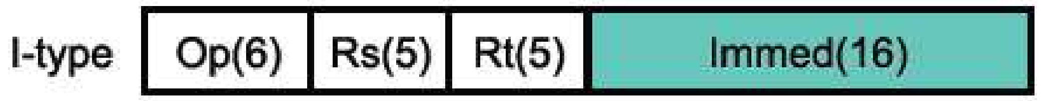


Number of Registers

- Registers faster than memory, have as many as possible?
  - **No**
- One reason registers are faster: there are **fewer of them**
  - Small is fast (hardware truism)
- Another: they are **directly addressed** (no address calc)
  - More registers, means more bits per register in instruction
  - Thus, fewer registers per instruction or larger instructions
- **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**
  - Across function calls, traps, and context switches
- Trend toward more registers:
  - 8 (x86) → 16 (x86-64), 16 (ARM v7) → 32 (ARM v8)

- **Addressing mode:** way of specifying address
  - Used in memory-memory or load/store instructions in register ISA
- Examples
  - **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?

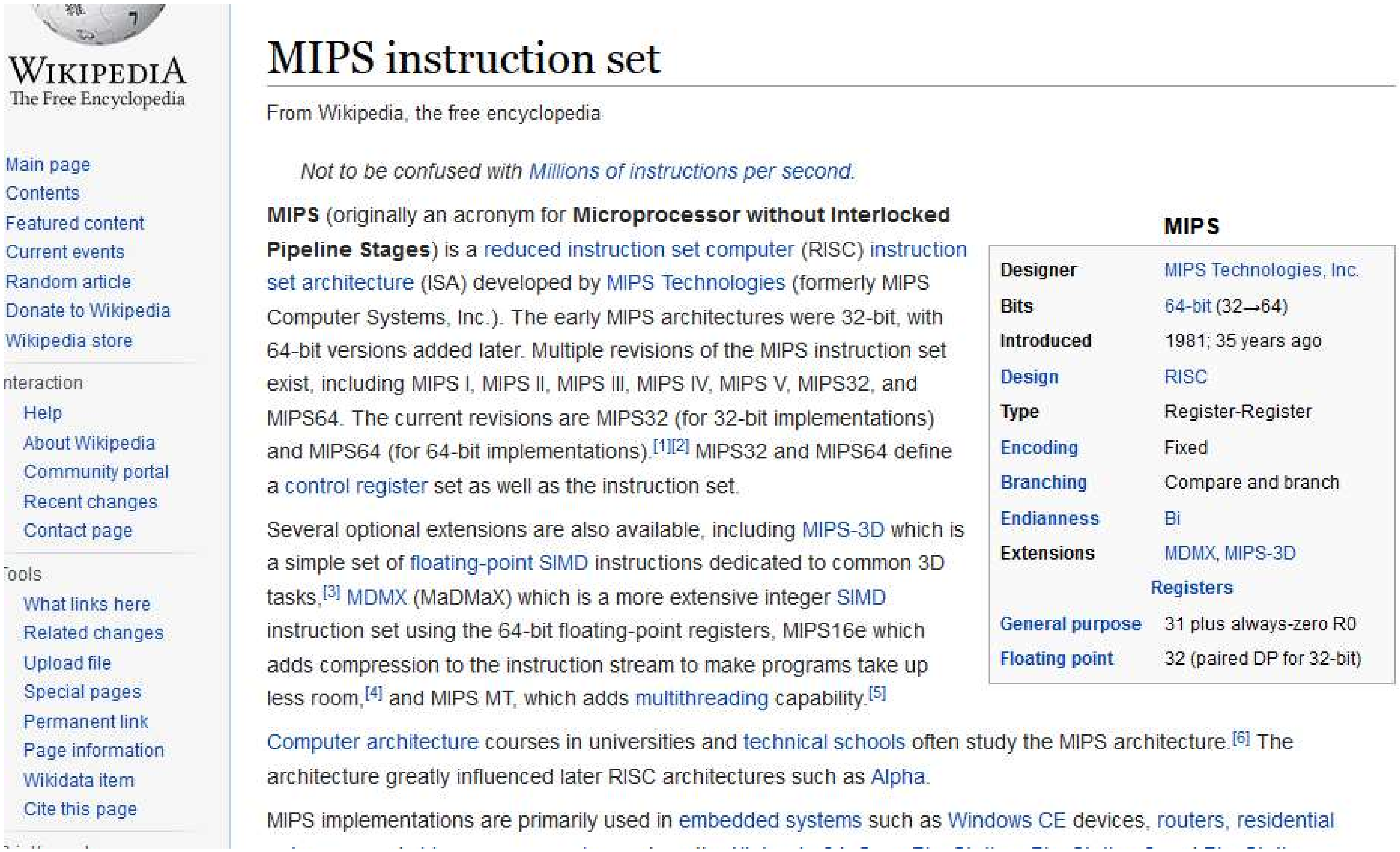
ADD R1, R2, #4



- MIPS
  - **Displacement**: R1+offset (16-bit)
  - Why? Experiments on VAX (ISA with every mode) found:
    - 80% use small displacement (or displacement of zero)
    - Only 1% accesses use displacement of more than 16bits

- Other ISAs (SPARC, x86) have reg+reg mode, too
  - Impacts both implementation and insn count? (How?)

- x86 (MOV instructions)
  - **Absolute**: zero + offset (8/16/32-bit)
  - **Register indirect**: R1
  - **Displacement**: R1+offset (8/16/32-bit)
  - **Indexed**: R1+R2
  - **Scaled:** R1 + (R2*Scale) + offset(8/16/32-bit)   Scale = 1, 2, 4, 8



MIPS instruction set

From Wikipedia, the free encyclopedia

Not to be confused with Millions of instructions per second.

MIPS (originally an acronym for **Microprocessor without Interlocked Pipeline Stages**) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies (formerly MIPS Computer Systems, Inc.). The early MIPS architectures were 32-bit, with 64-bit versions added later. Multiple revisions of the MIPS instruction set exist, including MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS32, and MIPS64. The current revisions are MIPS32 (for 32-bit implementations) and MIPS64 (for 64-bit implementations).[1][2] MIPS32 and MIPS64 define a control register set as well as the instruction set.

Several optional extensions are also available, including MIPS-3D which is a simple set of floating-point SIMD instructions dedicated to common 3D tasks,[3] MDMX (MaDMaX) which is a more extensive integer SIMD instruction set using the 64-bit floating-point registers, MIPS16e which adds compression to the instruction stream to make programs take up less room,[4] and MIPS MT, which adds multithreading capability.[5]

MIPS

Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981; 35 years ago
Design	RISC
Type	Register-Register
Encoding	Fixed
Branching	Compare and branch
Endianness	Bi
Extensions	MDMX, MIPS-3D
Registers	
General purpose	31 plus always-zero R0
Floating point	32 (paired DP for 32-bit)

Computer architecture courses in universities and technical schools often study the MIPS architecture.[6] The architecture greatly influenced later RISC architectures such as Alpha.

MIPS implementations are primarily used in embedded systems such as Windows CE devices, routers, residential

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1` = `$s2` + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[`$s2`+20]=`$s1`;`$s1`=`0 or 1`	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	`$s1` = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` | `$s3`)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1` = `$s2` & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1` = `$s2` | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1`!= `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

mov r1, #0

cmp r2, r1

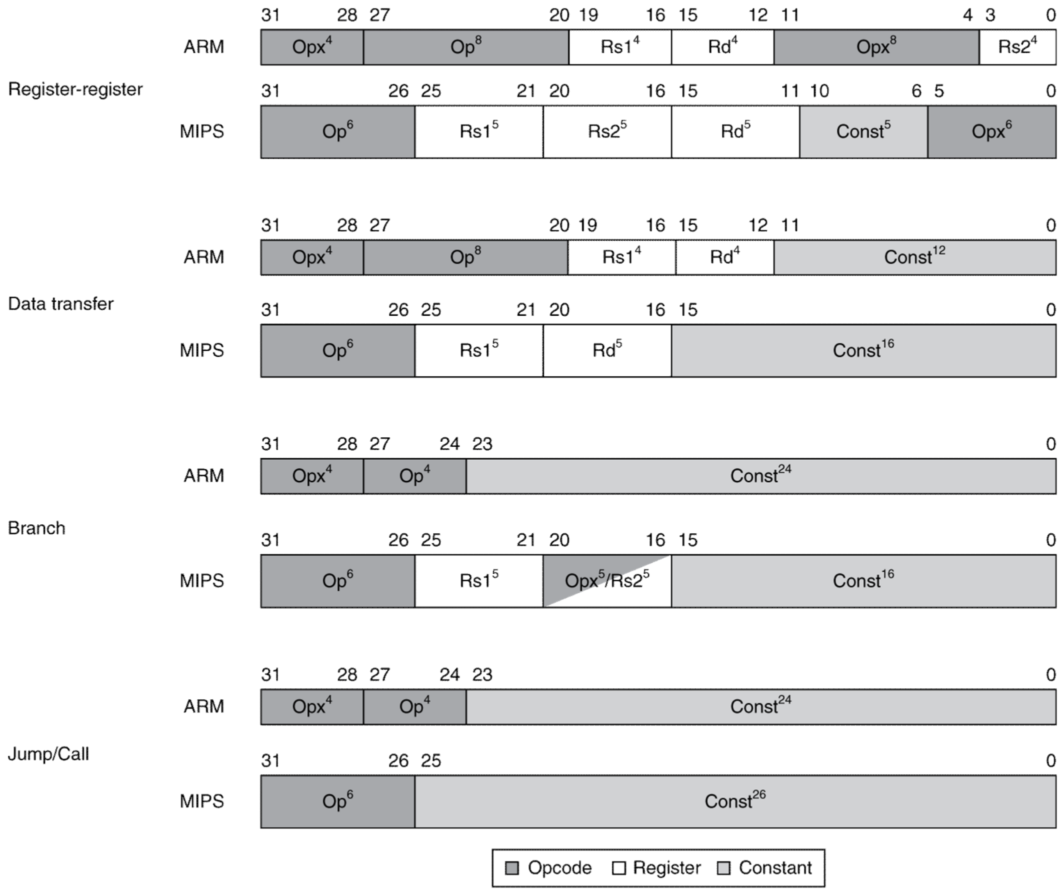
	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

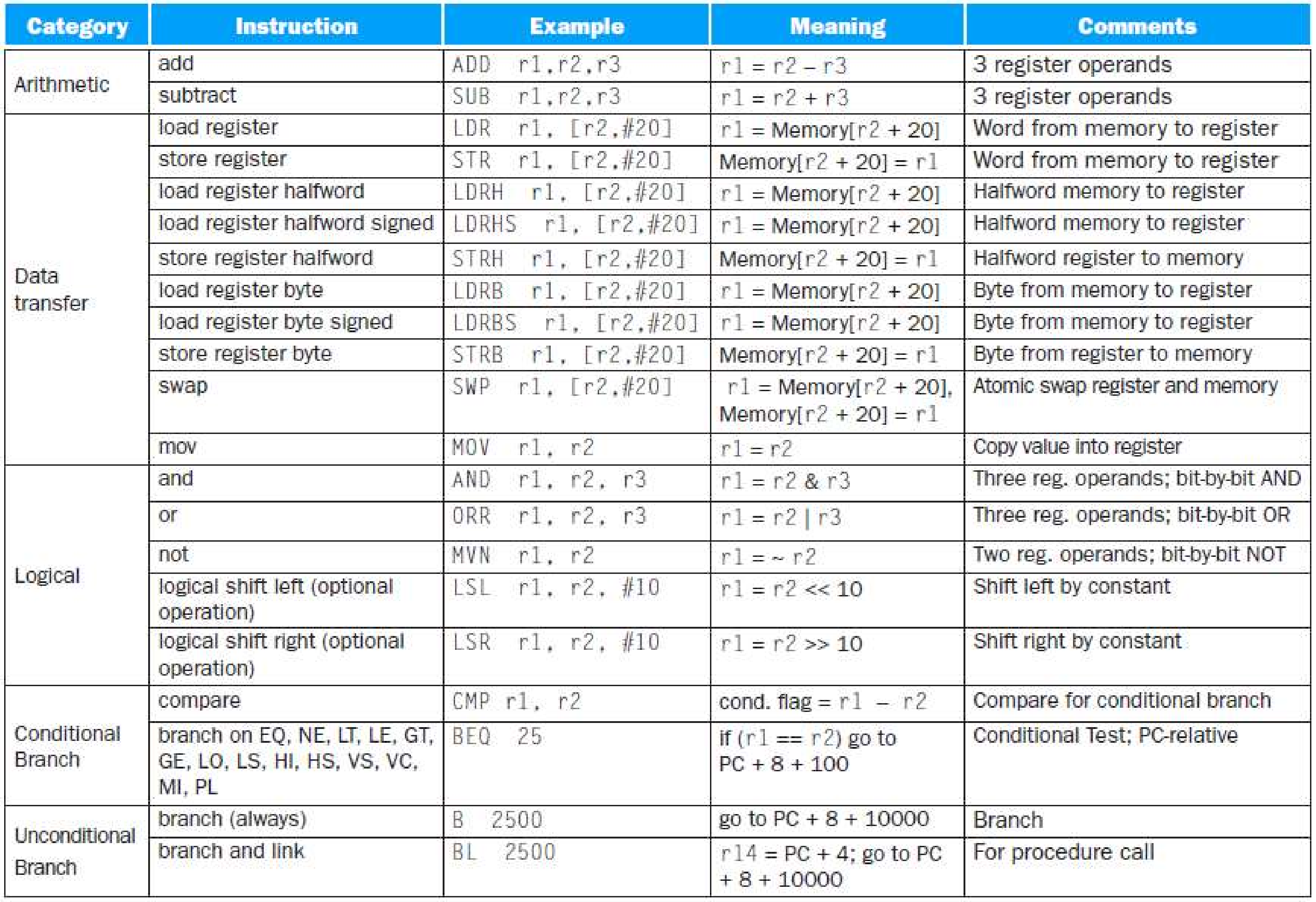
- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

ADD ADDEQ

Instruction Encoding



ARM assembly language



Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`ADD r1,r2,r3`	r1 = r2 – r3	3 register operands
	subtract	`SUB r1,r2,r3`	r1 = r2 + r3	3 register operands
Data transfer	load register	`LDR r1, [r2,#20]`	r1 = Memory[r2 + 20]	Word from memory to register
	store register	`STR r1, [r2,#20]`	Memory[r2 + 20] = r1	Word from memory to register
	load register halfword	`LDRH r1, [r2,#20]`	r1 = Memory[r2 + 20]	Halfword memory to register
	load register halfword signed	`LDRHS r1, [r2,#20]`	r1 = Memory[r2 + 20]	Halfword memory to register
	store register halfword	`STRH r1, [r2,#20]`	Memory[r2 + 20] = r1	Halfword register to memory
	load register byte	`LDRB r1, [r2,#20]`	r1 = Memory[r2 + 20]	Byte from memory to register
	load register byte signed	`LDRBS r1, [r2,#20]`	r1 = Memory[r2 + 20]	Byte from memory to register
	store register byte	`STRB r1, [r2,#20]`	Memory[r2 + 20] = r1	Byte from register to memory
	swap	`SWP r1, [r2,#20]`	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	Atomic swap register and memory
	mov	`MOV r1, r2`	r1 = r2	Copy value into register
Logical	and	`AND r1, r2, r3`	r1 = r2 & r3	Three reg. operands; bit-by-bit AND
	or	`ORR r1, r2, r3`	r1 = r2 | r3	Three reg. operands; bit-by-bit OR
	not	`MVN r1, r2`	r1 = ~ r2	Two reg. operands; bit-by-bit NOT
	logical shift left (optional operation)	`LSL r1, r2, #10`	r1 = r2 << 10	Shift left by constant
	logical shift right (optional operation)	`LSR r1, r2, #10`	r1 = r2 >> 10	Shift right by constant
Conditional Branch	compare	`CMP r1, r2`	cond. flag = r1 – r2	Compare for conditional branch
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	`BEQ 25`	if (r1 == r2) go to PC + 8 + 100	Conditional Test; PC-relative
Unconditional Branch	branch (always)	`B 2500`	go to PC + 8 + 10000	Branch
	branch and link	`BL 2500`	r14 = PC + 4; go to PC + 8 + 10000	For procedure call

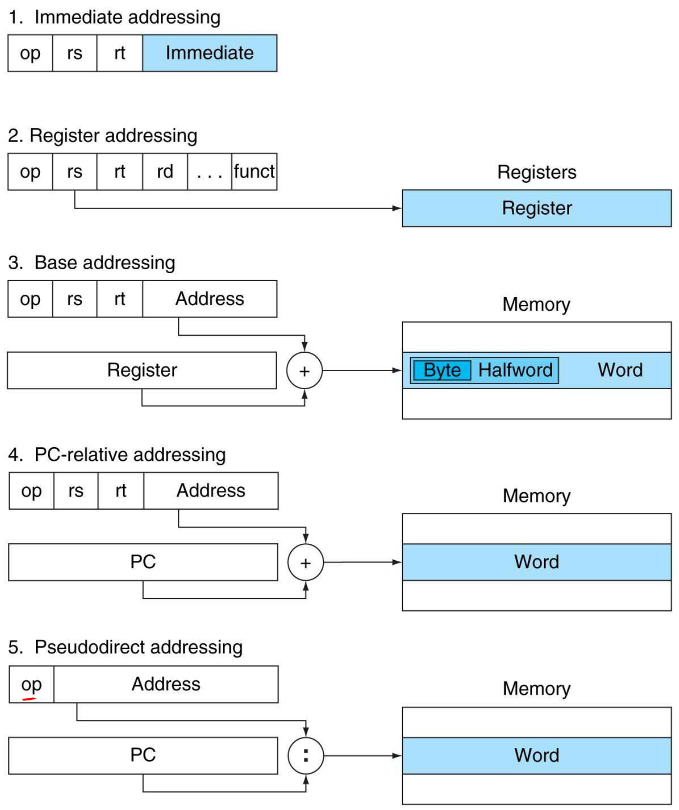


FIGURE 2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC. Copyright © 2009 Elsevier, Inc. All rights reserved.

The Intel x86 ISA

Evolution with backward compatibility

- 8080 (1974): 8-bit microprocessor
  - Accumulator, plus 3 index-register pairs
- 8086 (1978): 16-bit extension to 8080
  - Complex instruction set (CISC)
- 8087 (1980): floating-point coprocessor
  - Adds FP instructions and register stack
- 80286 (1982): 24-bit addresses, MMU
  - Segmented memory mapping and protection
- 80386 (1985): 32-bit extension (now IA-32)
  - Additional addressing modes and operations
  - Paged memory mapping as well as segments

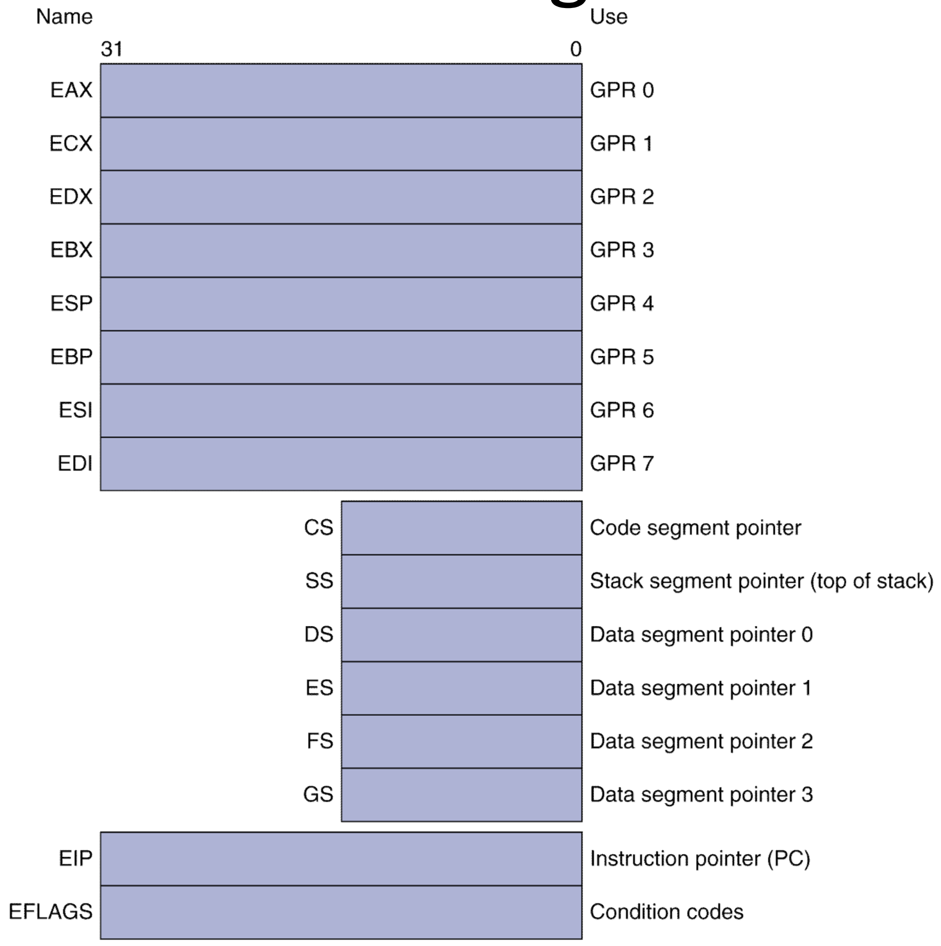
The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers



Basic x86 Addressing Modes

ARM : ADD r0 , r1 , r2

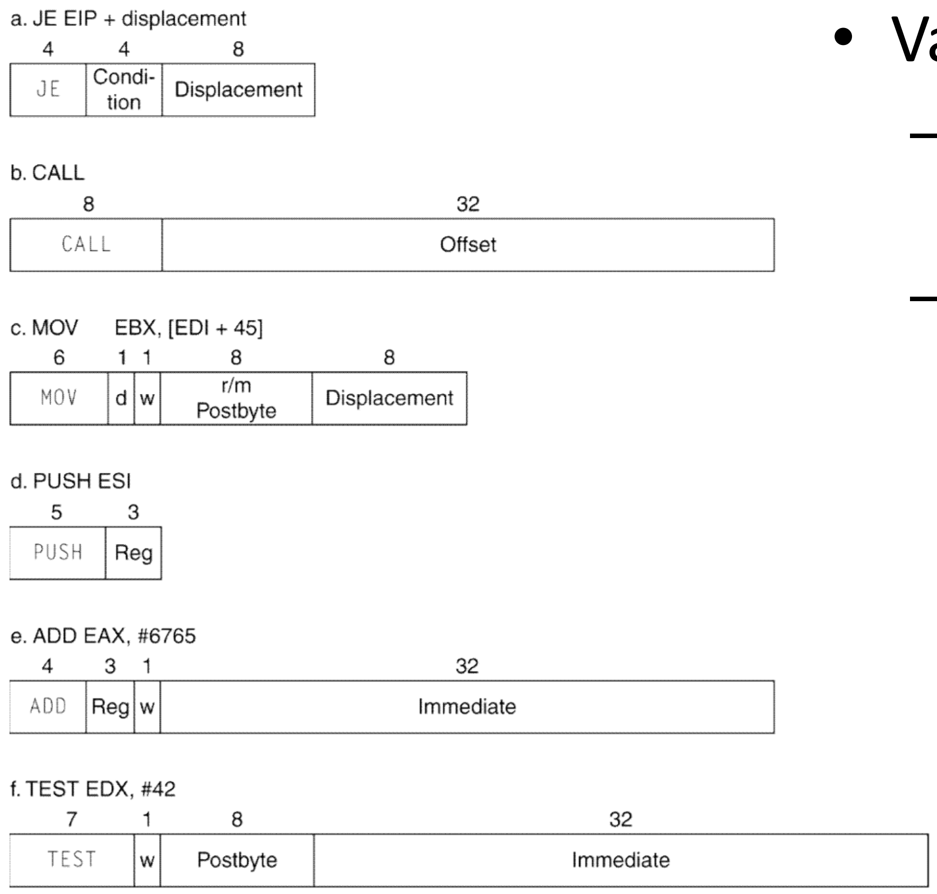
Intel: ADD r1 , r2

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

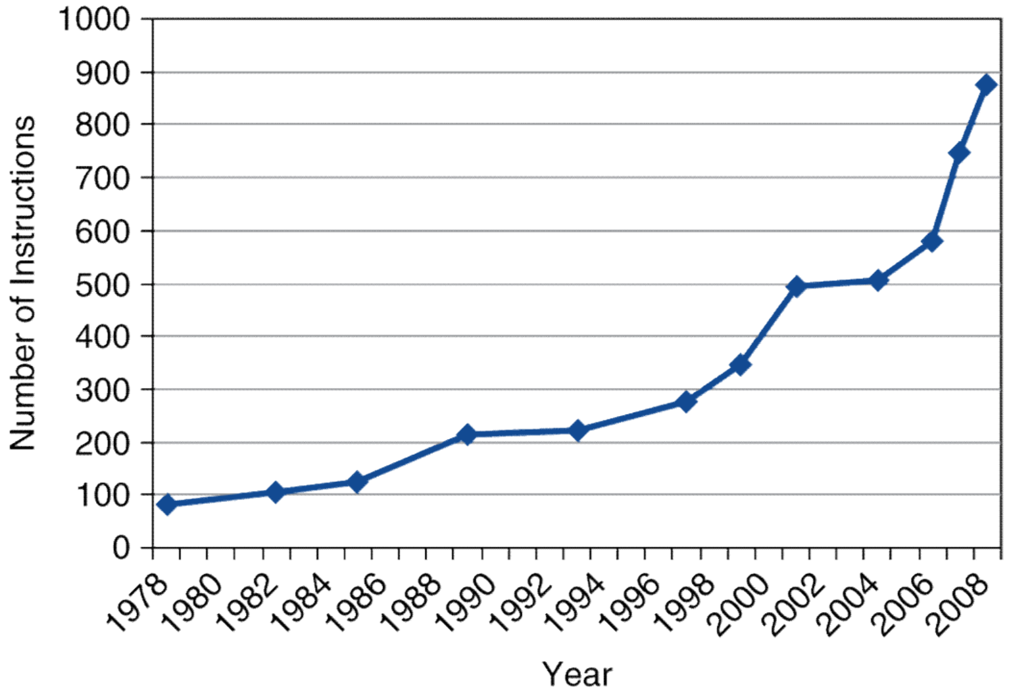
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

Fallacies

- Powerful instruction $\Rightarrow$ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Concluding Remarks

IC x CPI x CC

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

BL

- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%

In linking stage, does the label “else” in the following code require relocation or not?

```
BEQ   else
```

A: Yes
B: No