

Lectures on

Vol 3

Computer Architecture & ASSEMBLY Programming

By

Dr Jeff Drobman

website



drjeffsoftware.com/classroom.html

email



jeffrey.drobman@csun.edu

Index

Vol 3

❖ ISA & Assembly Code → 3

- MIPS → 5
- ARM → 44

❖ Old CISC MPU's → 72

❖ Interrupts → 79

❖ Other HW (I/O) → 103

❖ ICU → 116

- FSM
- uProgrammed
- Am2900 Bit-slice → 126

❖ RISC Pipelines → 137

- Clock Gen w/PLL → 144

❖ JTAG → 152

❖ JEDEC → 159

Section

ISA Assembly Code

Baseline Instruction Set

Rev Aug 2021

Computation

- ❖ ALU
 - ADD
 - SUB
 - AND
 - OR
 - XOR
 - NOT
- ❖ MULT/DIV [opt]
- ❖ BIT
 - SET/CLR
 - TEST
- ❖ COMPARE
 - CMP
- ❖ SHIFT
 - SHIFT (A, L)
 - ROTATE

Memory

- ❖ Reg-Reg
 - MOV
- ❖ Reg-Mem
 - LOAD
 - STORE
 - MOV
- ❖ Mem-Mem
 - MOV
- ❖ Stack
 - PUSH
 - POP

Program Control

- ❖ JUMP
 - JUMP/GOTO
- ❖ BRANCH
 - BRA
 - BRCC
 - LOOP
- ❖ CALL
 - CALL/CALR/JAL
 - RET/RETFIE
- ❖ NOP

I/O

- ❖ I/O
 - IN
 - OUT
- ❖ Mem Mapped
 - MOV PORT
 - LOAD/STORE

System Control

- ❖ Reset
 - RESET
- ❖ Power
 - SLEEP/HALT

RISC

CISC

OLD

NEW

MIPS

❖ MIPS I (32-bit) [R2000/3000]



❖ MIPS32 (32-bit, *MARS*)

❖ MIPS III (64-bit) [R4000]



❖ MIPS64 (64-bit)

- Superset of 32-bit ISA
- Adds 64-bit ops (“Double”)

➤ See separate slide set “MIPS”

MIPS microprocessors [\[edit \]](#)

The first MIPS microprocessor, the **R2000**, was announced in 1985. It added multiple-cycle multiply and divide instructions in a somewhat independent on-chip unit. New instructions were added to retrieve the results from this unit back to the register file; these result-retrieving instructions were interlocked.

The R2000 could be booted either **big-endian** or **little-endian**. It had thirty-one 32-bit general purpose registers, but no **condition code register** (the designers considered it a potential bottleneck), a feature it shares with the **AMD 29000** and the **Alpha**. Unlike other registers, the **program counter** is not directly accessible.

The R2000 also had support for up to four co-processors, one of which was built into the main CPU and handled exceptions, traps and memory management, while the other three were left for other uses. One of these could be filled by the optional **R2010 FPU**, which had thirty-two 32-bit registers that could be used as sixteen 64-bit registers for double-precision.

MIPS

MIPS Technologies

From Wikipedia, the free encyclopedia
(Redirected from [MIPS Computer](#))

MIPS Technologies, Inc., formerly **MIPS Computer Systems, Inc.**, was an [American](#) [fabless semiconductor design company](#) that is most widely known for developing the [MIPS architecture](#) and a series of [RISC CPU chips](#) based on it.^{[1][2]} MIPS provides [processor architectures](#) and cores for digital home, networking, embedded, [Internet of things](#) and mobile applications.^{[3][4]}

MIPS Technologies, Inc. is owned^[5] by Wave Computing, who acquired it from Tallwood MIPS Inc., a company indirectly owned by Tallwood Venture Capital. Tallwood bought it on 2017-10-25 from [Imagination Technologies](#), a [UK-based](#) company best known for their [PowerVR](#) graphics processor family.^[6] Imagination Technologies had previously bought MIPS after [CEVA, Inc.](#) pulled out of a bidding on 2013-02-08.

MIPS Technologies, Inc.



The former MIPS Technologies building in Santa Clara

Type	Subsidiary
Industry	RISC microprocessors
Fate	Acquired in 2018 by Wave Computing
Founded	1984; 36 years ago
Founder	John L. Hennessy 
Defunct	2013 
Headquarters	Sunnyvale, California, U.S.
Key people	Sandeep Vij
Products	Semiconductor intellectual property
Number of employees	up to 50 (according to LinkedIn in May 2018), previously 146 (September 2010)
Parent	Wave Computing 

History [\[edit \]](#)

MIPS Computer Systems Inc. was founded in 1984^{[7][8]} by a group of researchers from [Stanford University](#) that included [John L. Hennessy](#) and [Chris Rowen](#). These researchers had worked on a project called **MIPS** (for *Microprocessor without Interlocked Pipeline Stages*), one of the projects that pioneered the RISC concept. Other principal founders were Skip Stritter, formerly a Motorola technologist, and John Moussouris, formerly of IBM.^[9]

The initial CEO was Vaemon Crane, formerly President and CEO of [Computer Consoles Inc.](#), who arrived in February 1985 and departed in June 1989. He was replaced by Bob Miller, a former senior IBM and Data General executive. Miller ran the company through its IPO and subsequent sale to Silicon Graphics.

In 1988, MIPS Computer Systems designs were noticed by [Silicon Graphics](#) (SGI) and the company adopted the MIPS architecture for its computers.^[10] A year later, in December 1989, MIPS held its first **IPO**. That year, [Digital Equipment Corporation](#) (DEC) released a **Unix workstation** based on the MIPS design.

After developing the **R2000** and **R3000** microprocessors, a management change brought along the larger dreams of being a computer vendor. The company found itself unable to compete in the computer market against much larger companies and was struggling to support the costs of developing both the chips and the systems (**MIPS Magnum**). To secure the supply of future generations of MIPS microprocessors (the 64-bit **R4000**), SGI acquired the company in 1992^[11] for \$333 million^{[12][13]} and renamed it as MIPS Technologies Inc., a wholly owned subsidiary of SGI.^[14]

During SGI's ownership of MIPS, the company introduced the **R8000** in 1994 and the **R10000**^[15] in 1996 and a follow up the **R12000** in 1997.^[16] During this time, two future microprocessors code-named *The Beast* and *Capitan* were in development; these were cancelled after SGI decided to migrate to the **Itanium** architecture^[17] in 1998.^{[12][18]} As a result, MIPS was spun out as an intellectual property licensing company, offering licences to the MIPS architecture as well as microprocessor core designs.

Defunct	2013 
Headquarters	Sunnyvale, California, U.S.
Key people	Sandeep Vij
Products	Semiconductor intellectual property
Number of employees	up to 50 (according to LinkedIn in May 2018), previously 146 (September 2010)
Parent	Wave Computing 
Website	www.mips.com 

MIPS

Company timeline [\[edit \]](#)

Year ↕	↕
1981	Dr. John Hennessy at Stanford University founds and leads Stanford MIPS , a research program aimed at building a microprocessor using RISC principles.
1984	MIPS Computer Systems, Inc. co-founded by Dr. John Hennessy, Skip Stritter , and Dr. John Moussouris ^[43]
1986	First product ships: R2000 microprocessor, Unix workstation, and optimizing compilers
1988	R3000 microprocessor
1989	First IPO in November as MIPS Computer Systems with Bob Miller as CEO
1991	R4000 microprocessor
1992	SGI acquires MIPS Computer Systems. Transforms it into internal MIPS Group, and then incorporates and renames it to MIPS Technologies, Inc. (a wholly owned subsidiary of SGI)
1994	R8000 microprocessor
1994	Sony PlayStation released, using an R3000 CPU with custom GTE coprocessor
1996	R10000 microprocessor; Nintendo 64 released, incorporating a cut down R4300 processor.
1998	Re-IPO as MIPS Technologies, Inc
1999	Sony PlayStation 2 released, using an R5900 cpu with custom vector coprocessors
2002	Acquires Algorithmics Ltd, a UK-based MIPS development hardware/software and consultancy company.
September 6, 2005	Acquires First Silicon Solutions (FS2), a Lake Oswego, Oregon company as a wholly owned subsidiary. FS2 specializes in silicon IP, design services and OCI (On-Chip Instrumentation) development tools for programming, testing, debug and trace of embedded systems in SoC, SOPC, FPGA, ASSP and ASIC devices.
2007	MIPS Technologies acquires Portugal-based mixed-signal intellectual property company Chipidea
February 2009	MIPS Joins Linux Foundation ^[44]
May 8, 2009	Chipidea is sold to Synopsys .
June 2009	Android is ported to MIPS ^[45]

R3000: 32-bit CPU → *pipelined* (5 stages)

The **R3000** succeeded the R2000 in 1988, adding 32 KB (soon increased to 64 KB) caches for instructions and data, along with support for shared-memory **multiprocessing** in the form of a **cache coherence** protocol. While there were flaws in the R3000s multiprocessing support, it was successfully used in several successful multiprocessor computers. The R3000 also included a built-in **MMU**, a common feature on CPUs of the era. The R3000, like the R2000, could be paired with a **R3010** FPU. The R3000 was the first successful MIPS design in the marketplace, and eventually over one million were made. A speed-bumped version of the R3000 running up to 40 MHz, the **R3000A** delivered a performance of 32 **VUPs** (**VAX Unit of Performance**). The MIPS **R3000A**-compatible **R3051** running at 33.8688 MHz was the processor used in the **Sony PlayStation** though it didn't have FPU or MMU. Third-party designs include Performance Semiconductor's **R3400** and IDT's **R3500**, both of them were R3000As with an integrated R3010 FPU. **Toshiba's R3900** was a virtually first **SoC** for the early **handheld PCs** that ran **Windows CE**. A **radiation-hardened** variant for space applications, the **Mongoose-V**, is a R3000 with an integrated R3010 FPU.

The **R4000** series, released in 1991, extended MIPS to a full 64-bit architecture, moved the FPU onto the main die to create a single-chip microprocessor, and had a high clock frequency of 100 MHz at introduction. However, in order to achieve the clock frequency, the caches were reduced to 8 KB each and they took three cycles to access. The high operating frequencies were achieved through the technique of **deep pipelining** (called super-pipelining at the time). The improved **R4400** followed in 1993. It had larger 16 KB primary caches, largely bug-free 64-bit operation, and support for a larger L2 cache.

MIPS, now a division of SGI called MTI, designed the low-cost **R4200**, the basis for the even cheaper **R4300i**. A derivative of this microprocessor, the **NEC VR4300**, was used in the **Nintendo 64** game console.^[1]

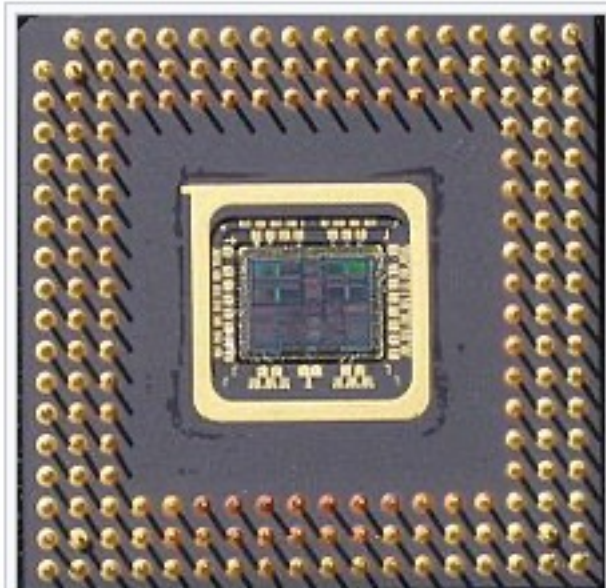
R4000: 1st 64-bit CPU → *super-pipelined* (8 stages)

Quantum Effect Devices (QED), a separate company started by former MIPS employees, designed the **R4600 Orion**, the **R4700 Orion**, the **R4650** and the **R5000**. Where the R4000 had pushed clock frequency and sacrificed cache capacity, the QED designs emphasized large caches which could be accessed in just two cycles and efficient use of silicon area.

MIPS

Wiki

R4700



Bottom-side view of package of R4700 Orion with the exposed silicon chip, fabricated by [IDT](#), designed by [Quantum Effect Devices](#)



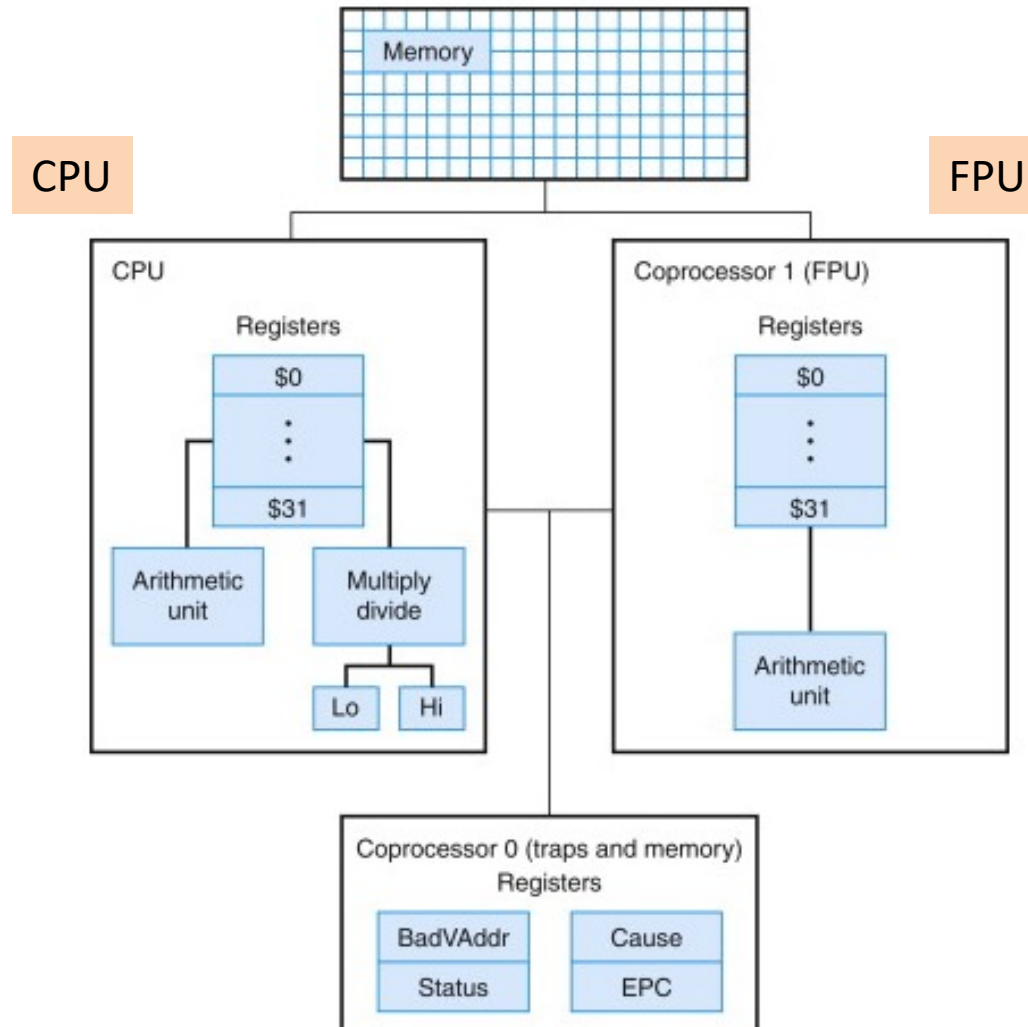
Top-side view of package for [R4700](#) Orion

MIPS I– Base (R2000) Org

Hennessy & Patterson

MIPS

Figure 7.10.1: MIPS R2000 CPU and FPU (COD
Figure A.10.1).



GP Registers

Register use convention:

Hennessy & Patterson

The calling convention described in this section is the one used by the gcc compiler. The native MIPS compiler uses a more complex convention that is slightly faster.

The MIPS CPU contains 32 general-purpose registers that are numbered 0–31. Register \$0 always contains the hardwired value 0.

- Registers \$at (1), \$k0 (26), and \$k1 (27) are reserved for the assembler and operating system and should not be used by user programs or compilers.
- Registers \$a0–\$a3 (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers \$v0 and \$v1 (2, 3) are used to return values from functions.
- Registers \$t0–\$t9 (8–15, 24, 25) are *caller-saved registers* that are used to hold temporary quantities that need not be preserved across calls (see COD Section 2.8 (Supporting Procedures in Computer Hardware)).
- Registers \$s0–\$s7 (16–23) are *callee-saved registers* that hold long-lived values that should be preserved across calls.
- Register \$gp (28) is a global pointer that points to the middle of a 64K block of memory in the static data segment.
- Register \$sp (29) is the stack pointer, which points to the last location on the stack. Register \$fp (30) is the frame pointer. The jal instruction writes register \$ra (31), the return address from a procedure call. These two registers are explain in COD Section A.7 (Exceptions and interrupts)

- ❖ \$a(0:3) *args*
- ❖ \$at, \$k(0:1) *reserved*
- ❖ \$v(0:1) *values*
- ❖ \$t(0-9) *temp*
- ❖ \$s(0:7) *saved*
- ❖ \$gp *global ptr*
- ❖ \$sp *stack ptr*
- ❖ \$fp *frame ptr*
- ❖ \$ra *return addr*


Instruction Formats

Wikipedia

MIPS

The following are the three formats used for the core instruction set:

Type	-31-	format (bits)					-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)	
I	opcode (6)	rs (5)	rt (5)	immediate (16)			
J	opcode (6)	address (26)					



R_d

R_{S1}

R_{S2}

- ☐ “shamt” ::= shift amount (5 bits)
- ☐ “funct” ::= function (opcode extension – 6 bits)

R, I Formats

MARS

R

add \$16,\$9,\$10

R_{S2}

18: add \$s0, \$t1, \$t2

R_d

R_{S1}

R_{S2}

$\$s0 = \$t1 + \$t2$

I

addi \$10,\$10,0x0000... 17: add \$t2, \$t2, 4

Im

R_d

R_{S1}

Im

\$t0	8	0x00000000
\$t1	9	0x6f57206f
\$t2	10	0x10040004
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x7f5b2073

R_d

Instruction Formats R & I

Hennessy & Patterson

PARTICIPATION
ACTIVITY

2.5.6: MIPS R-type and I-type instruction encoding (COD Figure 2.5).

Start

☐ 2x speed

Instruction formats

Instruction	Format
add	R
sub (subtract)	R
add immediate	I
lw (load word)	I
sw (store word)	I

			R _d		
op	rs	rt	rd	shamt	func
			(R-type)		
			Constant or address		
			(I-type)		
0	reg	reg	reg	0	32
0	reg	reg	reg	0	34
8	reg	reg	constant		
35	reg	reg	address		
43	reg	reg	address		

Sample instructions

add \$t0, \$s2, \$s3
sub \$s1, \$s2, \$s3
addi \$s1, \$s2, 20
lw \$t0, 1200(\$t1)
sw \$t0, 1200(\$t1)

	\$s2	\$s3	R _d =\$t0/\$s1		
0	18	19	8	0	32
0	18	19	17	0	34
8	18	\$s2	17	\$s1	20
35	9	8	1200		
43	9	\$t1	8	\$t0	1200

Register locations

\$t0 8
\$t1 9
\$s1 17
\$s2 18
\$s3 19

Address Formats

COMP122

Hennessy & Patterson

Format		Address computation
(register)	(\$at)	contents of register
imm	+4	immediate
imm (register)	+4 (\$at)	immediate + contents of register
label		address of label
label ± imm	Label +4	address of label + or – immediate
label ± imm (register)		address of label + or – (immediate + contents of register)

Label +4 (\$at)

“EA” (effective address)

- ❖ 1 component
 - Register (R)
 - Immediate (I)
 - Label (L)
- ❖ 2 components
 - R + I
 - L + I
- ❖ 3 components
 - R + L + I

Size of Immediate matters!

Simple ALU Code

Hennessy & Patterson

Table 1.5.1: Sample processor instructions

#Immediate data

Add X, #num, Y	Adds data in memory location <i>X</i> to the number <i>num</i> , storing result in location <i>Y</i>
Sub X, #num, Y	Subtracts <i>num</i> from data in location <i>X</i> , storing result in location <i>Y</i>
Mul X, #num, Y	Multiplies data in location <i>X</i> by <i>num</i> , storing result in location <i>Y</i>
Div X, #num, Y	Divides data in location <i>X</i> by <i>num</i> , storing result in location <i>Y</i>
Jmp Z	Tells the processor that the next instruction to execute is in memory location <i>Z</i>

“Go To”

Simple ALU Code

COMP122

Hennessy & Patterson

PARTICIPATION
ACTIVITY

1.5.4: Processor executing instructions.

Start



2x speed

#Immediate data

0	Mul 97, #9, 98
1	Div 98, #5, 98
2	Add 98, #32, 99
3	Jmp 0
4	...

Memory Locations

96	??
97	20
98	180
99	68

Processor

Mul 97, #9, 98
20 * 9 --> 180
Next: 0

Simple ALU Code

Hennessy & Patterson

PARTICIPATION ACTIVITY

1.5.3: Memory stores instructions and data as 0s and 1s.

Machine code

Location	Memory	Meaning
0	011 1100001 001001 1100010	Mul 97, #9, 98
1	100 1100010 000101 1100010	Div 98, #5, 98
2	001 1100010 100000 1100011	Add 98, #32, 99
3	101 00000000000000000000	Jmp 0
4	??	
...		
96	??	
97	000000000000000000010100	20
98	??	
99	??	

Assembly code

Location	Memory
0	Mul 97, #9, 98
1	Div 98, #5, 98
2	Add 98, #32, 99
3	Jmp 0
4	??
...	
96	??
97	20
98	??
99	??

MIPS Assembly

Hennessy & Patterson

PARTICIPATION ACTIVITY

2.5.1: Example of translating a MIPS assembly instruction into a machine instruction.

Start

☐

2x speed

add \$t0 \$s1 \$s2

add	\$s1	\$s2	\$t0	unused	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

MIPS Assembly

Instruction formats

Instruction	Format	op	rs	rt	rd	shamt	funct	(R-type)
					Constant or address			(I-type)
add	R	0	reg	reg	reg	0	32	
sub (subtract)	R	0	reg	reg	reg	0	34	
add immediate	I	8	reg	reg	constant			
lw (load word)	I	35	reg	reg	address			
sw (store word)	I	43	reg	reg	address			

Sample instructions

```
add $t0, $s2, $s3
sub $s1, $s2, $s3
addi $s1, $s2, 20
lw $t0, 1200($t1)
sw $t0, 1200($t1)
```

0	18	19	8	0	32
0	18	19	17	0	34
8	18	17	20		
35	9	8	1200		
43	9	8	1200		

Register locations

```
$t0    8
$t1    9
$s1   17
$s2   18
$s3   19
```

MIPS Assembly

Hennessy & Patterson

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17		100		addi \$s1,\$s2,100
lw	I	35	18	17		100		lw \$s1,100(\$s2)
sw	I	43	18	17		100		sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

offset

add \$t0 \$s1 \$s2

add	\$s1	\$s2	\$t0	unused	add
0	17	18	8	0	32

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

MIPS Assembly

Hennessy & Patterson

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multl	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
load half unsigned	lhu	I	branch greater than	bgt	I
store half	sh	I	branch greater than or equal	bge	I
load byte	lb	I			
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

Mult?
la

Multiply & Divide

MULTIPLY

- ❖ *Unsigned* only
- ❖ First convert negative numbers (2sC) – NEG op
- ❖ Compute result sign: 0 if both signs same, 1 else (not=)
- ❖ Complement result if sign is negative – NEG op
- ❖ Other MPUs use *signed* multiply (2sC) via “Booth’s Algorithm”

DIVIDE

- ❖ No hardware, no instruction
- ❖ Create subroutine (may find ones in asm library)
- ❖ Compute
 - Long division
 - Non-restoring division
 - Iterative subtraction (very slow)
- ❖ Use tricks
 - Divide by **2** or any **2ⁿ**: right SHIFT by n
 - Divide by **10**: convert to BCD, then right SHIFT by 4 (reconvert to binary)
 - Divide by **5**: divide by 10, then multiply by 2 (by shifting after conv. Bin)

Shift/Rotate & Bit

❖ SHIFT

❑ SHIFT

- Arithmetic
- Logical

❑ ROTATE

- Carry Bit (with, w/o)

❖ BIT

❑ SET

❑ CLR

❑ TEST

Figure 2.6.1: C and Java logical operators and their corresponding MIPS instructions (COD Figure 2.8).

MIPS implements NOT using a NOR with one operand being zero.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

Addressing Modes

Hennessy & Patterson

COMP122

MIPS addressing mode summary

Multiple forms of addressing are generically called *addressing modes*. The figure below shows how operands are identified for each addressing mode. The MIPS addressing modes are the following:

1. **Immediate addressing**: The operand is a constant within the instruction itself
2. **Register addressing**: The operand is a register
3. **Base addressing / displacement addressing**: The operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. **PC-relative addressing**: The branch address is the sum of the PC and a constant in the instruction
5. **Pseudodirect addressing**: The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

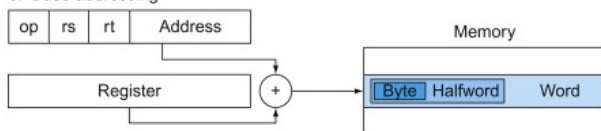
1. Immediate addressing



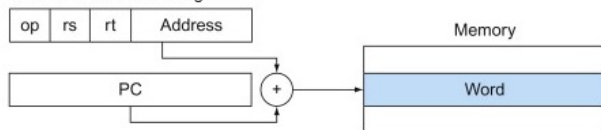
2. Register addressing



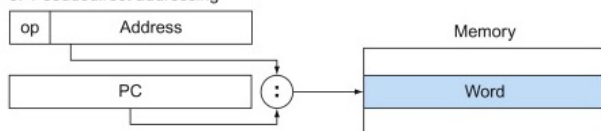
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Addressing Modes

Hennessy & Patterson

1. Immediate addressing



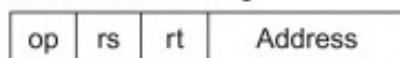
2. Register addressing



Registers

Register

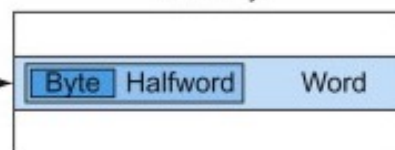
3. Base addressing



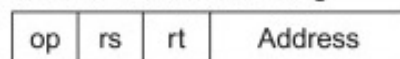
Memory



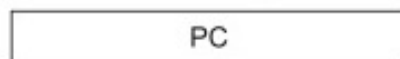
+



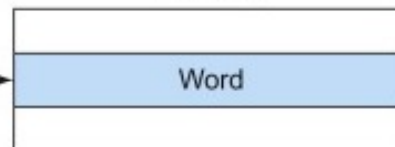
4. PC-relative addressing



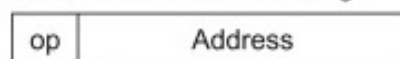
Memory



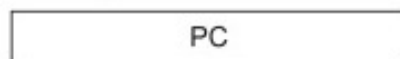
+



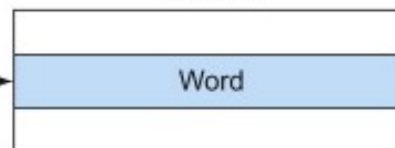
5. Pseudodirect addressing



Memory



:




Load *Pseudo* Ops

The MIPS assembler (and SPIM) synthesizes the more complex addressing modes by producing one or more instructions before the load or store to compute a complex address. For example, suppose that the label `table` referred to memory location `0x10000004` and a program contained the instruction

```
ld $a0, table + 4($a1)
```

The assembler would translate this instruction into the instructions



```
lui $at, 4096  
addu $at, $at, $a1  
lw $a0, 8($at)
```

The first instruction loads the upper bits of the label's address into register `$at`, which is the register that the assembler reserves for its own use. The second instruction adds the contents of register `$a1` to the label's partial address. Finally, the load instruction uses the hardware address mode to add the sum of the lower bits of the label's address and the offset from the original instruction to the value in register `$at`.

la = load address (32-bit)

Lab 1B: Load Addr

```
.eqv heapHi, 0x1004
```



```
lui $t3, heapHi
```

```
.eqv heap, 0x10040000
```



```
la $t3, heap
```

Immediates (16/32-bit)

1. Immediate addressing



32-bit immediate operands

PARTICIPATION
ACTIVITY

2.10.1: The lui instruction, and how to load a 32-bit constant (COD Figure 2.17 (The effect of the lui instruction)).

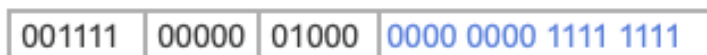
Start

☐ 2x speed

lui \$t0, 255 # \$t0 is register 8

lui

lui instruction:



How load following 32-bit constant into \$s0?

0000 0000 0011 1101 0000 1001 0000 0000
(61 in decimal) (2304 in decimal)

High HW

lui \$s0, 61

ori \$s0, \$s0, 2304

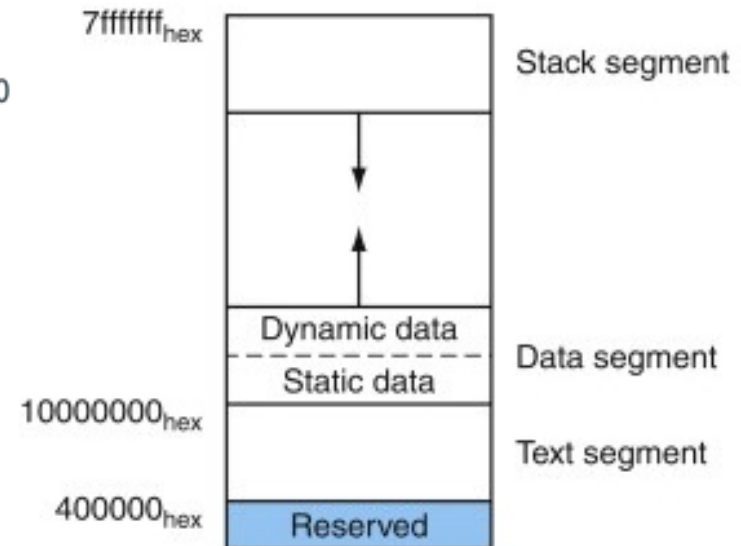
Low HW



Load: lui vs. \$gp

Because the data segment begins far above the program at address 10000000_{hex} , load and store instructions cannot directly reference data objects with their 16-bit offset fields (see COD Section 2.5 (Representing Instructions in the Computer)). For example, to load the word in the data segment at address 10010020_{hex} into register $\$v0$ requires two instructions:

```
lui $s0, 0x1001      # 0x1001 means 1001 base 16
lw $v0, 0x0020($s0)  # 0x10010000 + 0x0020 = 0x10010020
```



To avoid repeating the `lui` instruction at every load and store, MIPS systems typically dedicate a register ($\$gp$) as a *global pointer* to the static data segment. This register contains address 10008000_{hex} , so load and store instructions can use their signed 16-bit offset fields to access the first 64 KB of the static data segment. With this global pointer, we can rewrite the example as a single instruction:

```
lw $v0, 0x8020($gp)
```

Of course, a global pointer register makes addressing locations $10000000_{\text{hex}} - 10010000_{\text{hex}}$ faster than other heap locations. The MIPS compiler usually stores *global variables* in this area, because these variables have fixed locations and fit better than other global data, such as arrays.

Load Immediate

li vs. la

```

28 #short/long immediates
29 li $t1, 256 #short I
30 li $t2, -2 #short neg
31 li $t3, 0x12345678 #long I
32 la $t4, 0x12345678 #same?
33 add $t5,$t1,$t2 #R format
34 addi $t5,$t2, 3300 #I format--too large?

```

Code	Basic	Source
0xad6c0008	sw \$12,0x00000008(\$11)	27: sw \$t4, 8(\$t3)
0x24090100	addiu \$9,\$0,0x00000100	29: li \$t1, 256 #short I
0x240afffe	addiu \$10,\$0,0xffff...	30: li \$t2, -2 #short neg
0x3c011234	lui \$1,0x00001234	31: li \$t3, 0x12345678 #long I
0x342b5678	ori \$11,\$1,0x00005678	
0x3c011234	lui \$1,0x00001234	32: la \$t4, 0x12345678 #same?
0x342c5678	ori \$12,\$1,0x00005678	
0x012a6820	add \$13,\$9,\$10	33: add \$t5,\$t1,\$t2 #R format
0x214d0ce4	addi \$13,\$10,0x0000...	34: addi \$t5,\$t2, 3300 #I format--too large?

Loads (ea)

MARS

```
lui $1,0x00001001    12: lw $t1, hello
lw $9,0x00000000($1)
```

Address	Code	Basic	Source
0x00400000	0x24090004	addiu \$9,\$0,0x00000004	10: li \$t1, 4 #next word
0x00400004	0x3c011001	lui \$1,0x00001001	11: la \$t2, data #data
0x00400008	0x342a0000	ori \$10,\$1,0x00000000	
0x0040000c	0x3c011004	lui \$1,0x00001004	12: li \$t3, 0x10041111 #heap
0x00400010	0x342b1111	ori \$11,\$1,0x00001111	
0x00400014	0x8d510000	lw \$17,0x00000000(\$10)	14: lw \$s1, (\$t2) #aaaa
0x00400018	0x8d52000c	lw \$18,0x0000000c(\$10)	15: lw \$s2, 12(\$t2) #dddd
0x0040001c	0x3c011001	lui \$1,0x00001001	16: lw \$s3, data+4 #bbbb
0x00400020	0x8c330004	lw \$19,0x00000004(\$1)	
0x00400024	0x3c011001	lui \$1,0x00001001	17: lw \$s4, data+4(\$t1) #cccc
0x00400028	0x00290821	addu \$1,\$1,\$9	
0x0040002c	0x8c340004	lw \$20,0x00000004(\$1)	

EA (Eff Addr) Formats

MARS

MIPS

MARS

License

Bugs/Comments

Acknowledgements

Instruction Set Song

100

unsigned 16-bit integer (0 to 65535)

100000

signed 32-bit integer (-2147483648 to 2147483647)

Load & Store addressing mode, basic instructions

-100(\$t2)

sign-extended 16-bit integer added to contents of \$t2

Load & Store addressing modes, pseudo instructions

(\$t2)

contents of \$t2

-100

signed 16-bit integer

100

unsigned 16-bit integer

100000

signed 32-bit integer

100(\$t2)

zero-extended unsigned 16-bit integer added to contents of \$t2

100000(\$t2)

signed 32-bit integer added to contents of \$t2

label

32-bit address of label

label(\$t2)

32-bit address of label added to contents of \$t2

label+100000

32-bit integer added to label's address

label+100000(\$t2)

sum of 32-bit integer, label's address, and contents of \$t2

Basic Instructions

Extended (pseudo) Instructions

Directives

Syscalls

Exceptions

M...

Load Formats

COMP122

MARS

<code>li \$t1,-100</code>	Load Immediate : Set \$t1 to 16-bit immediate (sign-extended)
<code>li \$t1,100</code>	Load Immediate : Set \$t1 to unsigned 16-bit immediate (zero-extended)
<code>li \$t1,100000</code>	Load Immediate : Set \$t1 to 32-bit immediate

<code>lw \$t1,-100(\$t2)</code>	Load word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,(\$t2)</code>	Load Word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,-100</code>	Load Word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,100</code>	Load Word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,100000</code>	Load Word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,100(\$t2)</code>	Load Word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,100000(\$t2)</code>	Load Word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,label</code>	Load Word : Set \$t1 to contents of memory word at label's address
<code>lw \$t1,label(\$t2)</code>	Load Word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,label+100000</code>	Load Word : Set \$t1 to contents of effective memory word address
<code>lw \$t1,label+100000(\$t2)</code>	Load Word : Set \$t1 to contents of effective memory word address

Store Formats

MARS

sw \$t1,-100(\$t2)

Store word : Store contents of \$t1 into effective memory word address

sw \$t1,(\$t2)

Store Word : Store \$t1 contents into effective memory word address

sw \$t1,-100

Store Word : Store \$t1 contents into effective memory word address

sw \$t1,100

Store Word : Store \$t1 contents into effective memory word address

sw \$t1,100000

Store Word : Store \$t1 contents into effective memory word address

sw \$t1,100(\$t2)

Store Word : Store \$t1 contents into effective memory word address

sw \$t1,100000(\$t2)

Store Word : Store \$t1 contents into effective memory word address

sw \$t1,label

Store Word : Store \$t1 contents into memory word at label's address

sw \$t1,label(\$t2)

Store Word : Store \$t1 contents into effective memory word address

sw \$t1,label+100000

Store Word : Store \$t1 contents into effective memory word address

sw \$t1,label+100000(\$t2)

Store Word : Store \$t1 contents into effective memory word address

Control

❖ Control Structures

☐ IF-THEN-ELSE

☐ LOOPS

☐ Subroutines/Functions → *Methods*

Assembly level uses

➤ Conditional **Branches (B)**

Assembly level uses

➤ **Jump and Link (JAL)**

❖ Modern CPU's use

➤ “branch prediction” for

➤ “speculative execution”

Conditionals

High level

IF-THEN-ELSE

Use **conditions**: *logical* expressions (T/F)

❖ Assembly uses conditionals too

- ☐ All use *conditional **Branches*** (**B**<cond>)
- ☐ **Flags** must be set first (T/F): **N, Z** for {< = >}
 - Use any ALU op, or “Compare” (cmp)
 - ARM must use “Compare” (cmp)
- ☐ ARM (only) supports conditionals for ALL ops
 - Add<cond> → Add gtz

Conditionals

BNF

$\langle \text{conditional} \rangle ::= \langle \text{logic exp} \rangle$

$\langle \text{logic exp} \rangle ::= [\langle \text{Bool} \rangle] \langle \text{logic op} \rangle \langle \text{Bool} \rangle \mid \langle \text{logic exp} \rangle$

$\langle \text{logic op} \rangle ::= \{\text{AND, OR, NOT, XOR, NAND, NOR, XNOR}\}$

$\langle \text{Bool} \rangle ::= \{\text{true, false}\} \mid \langle \text{relational exp} \rangle$

$\langle \text{relational exp} \rangle ::= \langle \text{arith val} \rangle \langle \text{rel op} \rangle \langle \text{arith val} \rangle \mid \langle \text{rel exp} \rangle$

$\langle \text{rel op} \rangle ::= \{<, =, !=, >, <=, >=\}$

$\langle \text{rel op} \rangle ::= \{\text{lt, **eq**, **ne**, gt, gte, lte}\} \mid \langle \text{pseudo rel_Z} \rangle$

$\langle \text{pseudo rel_Z} \rangle ::= \{\text{ltz, ez, nz, gtz, gtez, ltez}\}$

Z → Uses \$0 as 2nd operand

Jump/Branch

Program Control

❖ JUMP

- JUMP/GOTO

➤ Long address (absolute pointer)

☐ Used for ***Swap/task switching***

❖ BRANCH

- BRA
- BRCC
- LOOP

➤ PC relative (offset)

☐ Used for ***Loops***

❖ CALL

- CALL/CALR
- RET/RETFIE

➤ Absolute/relative address

☐ Used for ***Subroutines***

☐ ***PC ↔ SP***

☐ Used for ***Delays***

❖ NOP

❖ Absolute vs. Relative (PC)

❖ **Conditional** vs. ***Un*-conditional**

Control CP0

7.7 Exceptions and interrupts

Hennessy & Patterson

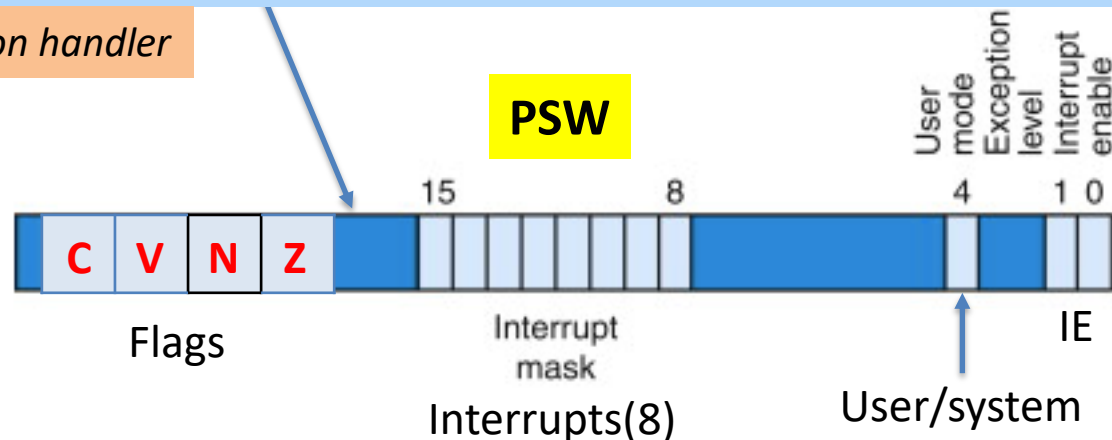
Figure 7.7.1: Coprocessor 0 registers.

Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

Figure 7.7.2: The status register (COD Figure

Interrupt handler: A piece of code that is run as a result of an exception or an interrupt.

Exception handler



Exceptions (EPC)

Hennessy & Patterson

Figure 7.7.3: The cause register (COD Figure A.7.2).

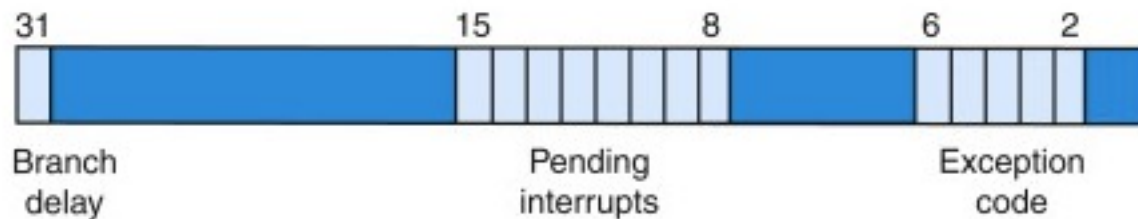


Figure 7.7.4: Causes of exceptions.

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	Ri	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

ISA

ARM

❖ v5 (32-bit, *ARMsim*)

➔ ❖ v7 (32-bit)

❖ v8 (64-bit)

- Superset of 32-bit ISA
 - Adds 64-bit ops (“Double”)
 - Simplifies ISA (more like MIPS)
- See separate slide set “ARM”

ISA: MIPS vs ARM

Instruction set: The vocabulary of commands understood by a given architecture.

1. ARMv7 is similar to MIPS. More than 9 billion chips with ARM processors were manufactured in 2011, making it the most popular instruction set in the world.
2. The second example is the Intel x86, which powers both the PC and the cloud of the PostPC Era.
3. The third example is ARMv8, which extends the address size of the ARMv7 from 32 bits to 64 bits. Ironically, as we shall see, this 2013 instruction set is closer to MIPS than it is to ARMv7.

❖ MIPS32

❖ ARMv7

32-bit

❖ MIPS64

❖ ARMv8

64-bit

ISA: MIPS vs ARM

MIPS

ARM

❖ CONDITIONALS

Branches only

ANY op

❖ LOAD/STORE

- Extras
- Memory Refs (EA)

LUI, LWL/R

LDM/STM

❖ SYSTEM

Break
Syscall

BKPT
DBG
HLT
Syscall?

❖ MISC Extras

“Q” (ALU ops)
PUSH/POP

Loads: MIPS vs ARM

MIPS

- ❖ MIPS
- ❖ lw, lh, lb are *primitives*
- ❖ "lui" is a primitive (load "upper" half of register)
- ❖ "li" and "la" are *pseudo-ops*
- ❖ lwl, lwr are *primitives*

- lw
- li
- la

ARM

- ❖ ARM
- ❖ most ARM Loads are *primitives*
- ❖ "B/H" width modifiers too: ldrb, ldrh
- ❖ ARM has an additional "Load Multiple" and "swap" as *primitives*
- ❖ ARM v7 ISA:
- ❖ ldr [condition](b, h, w)[s]
- ❖ str [condition](b, h, w)[s]
- ❖ ldm [load multiple]
- ❖ swp (b, h, w) [swap]

- Ldr
- Ldr #
- Ldr =

ARM Assembly



ARM Ref

Branch/jump:

B{cond}

BNE, BEQ (also B uncond.)

BL{cond}

Branch & Link

<no J>

returns:

ERET

conditionals:

IT (if-then)

debug:

BKPT

DBG (debug)

HLT (halt)

ARM Registers

ARMv7

Table A1-2 Predeclared core registers in AArch32 state

MIPS

\$t0-9, \$s0-7

\$a0-3

\$v0-1

\$gp

\$sp

\$ra

N/A

Register names	Meaning
r0-r15 and R0-R15	General purpose registers.
a1-a4	Argument, result or scratch registers. These are synonyms for R0 to R3.
v1-v8	Variable registers. These are synonyms for R4 to R11.
SB	Static base register. This is a synonym for R9.
IP	Intra-procedure call scratch register. This is a synonym for R12.
SP	Stack pointer. This is a synonym for R13.
LR	Link register. This is a synonym for R14.
PC	Program counter. This is a synonym for R15.

With the exception of a1-a4 and v1-v8, you can write the register names either in all upper case or all lower case.

ARM Registers – ARMSim

ARMv5

```
R0      : 000010a0
R1      : 00000001
R2      : 65707954
R3      : 00004014
R4      : 00000037
R5      : 00000002
R6      : 00000000
R7      : 00000000
R8      : 00000000
R9      : 00000000
R10 (s1): 00000000
R11 (fp): 00000000
R12 (ip): 00000000
R13 (sp): 00011400
R14 (lr): 00001024
R15 (pc): 00001058
```

PSW

```
-----
CPSR Register
Negative (N) : 0
Zero (Z)     : 1
Carry (C)    : 1
Overflow (V) : 0
IRQ Disable  : 1
FIQ Disable  : 1
Thumb (T)    : 0
CPU Mode     : System
-----
0x600000df
```

Thumb mode
CPU mode



ARM Registers

Hennessy & Patterson

ARMv8

Name	Register number	Usage	Preserved on call?
X0-X7	0-7	Arguments/Results	no
X8	8	Indirect result location register	no
X9-X15	9-15	Temporaries	no
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	no
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	no
X18	18	Platform register for platform independent code; otherwise a temporary register	no
X19-X27	19-27	Saved \$s0-7	yes
X28 (SP)	28	Stack Pointer \$sp	yes
X29 (FP)	29	Frame Pointer \$fp	yes
X30 (LR)	30	Link Register (return address) \$ra	yes
XZR	31	The constant value 0 \$zero	n.a.

ARM: Status & Control

CPSR

PSW

ARMv7

The Current Program Status Register (CPSR) has the following 32 bits.¹

- M (bits 0–4) is the processor mode bits.
- T (bit 5) is the Thumb state bit.
- F (bit 6) is the FIQ disable bit.
- I (bit 7) is the IRQ disable bit.
- A (bit 8) is the imprecise data abort disable bit.
- E (bit 9) is the data endianness bit.
- IT (bits 10–15 and 25–26) is the if-then state bits.
- GE (bits 16–19) is the greater-than-or-equal-to bits.
- DNM (bits 20–23) is the do not modify bits.
- J (bit 24) is the Java state bit.
- Q (bit 27) is the sticky overflow bit.
- V (bit 28) is the overflow bit.
- C (bit 29) is the carry/borrow/extend bit.
- Z (bit 30) is the zero bit.
- N (bit 31) is the negative/less than bit.

Flags

ARM Assembly

COMP122 ARM v7 ISA

ARM Ref

Load/store:

LDR (b, h, w)

STR (b, h, w)

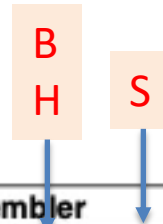
LDM{IA} [load multiple]

STM [store multiple]

SWP ((b, w) [swap]

PUSH/POP

ARM Instruction Set Quick Reference Card



Single data item loads and stores		§	Assembler
Load or store word, byte or halfword	Immediate offset		<op>{size}{T} Rd, [Rn {, #<offset>}](!)
	Post-indexed, immediate		<op>{size}{T} Rd, [Rn], #<offset>
	Register offset		<op>{size} Rd, [Rn, +/-Rm {, <opsh>}](!)
	Post-indexed, register		<op>{size}{T} Rd, [Rn], +/-Rm {, <opsh>}
	PC-relative		<op>{size} Rd, <label>
Load multiple	Block data load		LDM{IA IB DA DB} Rn(!), <reglist-PC>
	return (and exchange)		LDM{IA IB DA DB} Rn(!), <reglist+PC>
	and restore CPSR		LDM{IA IB DA DB} Rn(!), <reglist+PC>^
	User mode registers		LDM{IA IB DA DB} Rn. <reglist-PC>^
Push			PUSH <reglist>
Pop			POP <reglist>

ARM Quick Ref

COMP122

ARM Instruction Set Quick Reference Card

ARM architecture versions	
<i>n</i>	ARM architecture version <i>n</i> and above
<i>n</i> T, <i>n</i> J	T or J variants of ARM architecture version <i>n</i> and above
5E	ARM v5E, and 6 and above
T2	All Thumb-2 versions of ARM v6 and above
6K	ARMv6K and above for ARM instructions, ARMv7 for Thumb
Z	All Security extension versions of ARMv6 and above
RM	ARMv7-R and ARMv7-M only
XS	XScale coprocessor instruction

Flexible Operand 2		
Immediate value	#<imm8m>	
Register, optionally shifted by constant (see below)	Rm {, <opsh>}	
Register, logical shift left by register	Rm, LSL Rs	
Register, logical shift right by register	Rm, LSR Rs	
Register, arithmetic shift right by register	Rm, ASR Rs	
Register, rotate right by register	Rm, ROR Rs	

Register, optionally shifted by constant		
(No shift)	Rm	Same as Rm, LSL #0
Logical shift left	Rm, LSL #<shift>	Allowed shifts 0-31
Logical shift right	Rm, LSR #<shift>	Allowed shifts 1-32
Arithmetic shift right	Rm, ASR #<shift>	Allowed shifts 1-32
Rotate right	Rm, ROR #<shift>	Allowed shifts 1-31
Rotate right with extend	Rm, RRX	

PSR fields (use at least one suffix)		
Suffix	Meaning	
c	Control field mask byte	PSR[7:0]
f	Flags field mask byte	PSR[31:24]
s	Status field mask byte	PSR[23:16]
x	Extension field mask byte	PSR[15:8]

Condition Field		
Mnemonic	Description	Description (VFP)
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS / HS	Carry Set / Unsigned higher or same	Greater than or equal, or unordered
CC / LO	Carry Clear / Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

All ARM instructions (except those with Note C or Note U) can have any one of these condition codes after the instruction mnemonic (that is, before the first space in the instruction as shown on this card). This condition is encoded in the instruction.

All Thumb-2 instructions (except those with Note U) can have any one of these condition codes after the instruction mnemonic. This condition is encoded in a preceding IT instruction (except in the case of conditional Branch instructions). Condition codes in instructions must match those in the preceding IT instruction.

On processors without Thumb-2, the only Thumb instruction that can have a condition code is B <label>.

Processor Modes	
16	User
17	FIQ Fast Interrupt
18	IRQ Interrupt
19	Supervisor
23	Abort
27	Undefined
31	System

Prefixes for Parallel Instructions	
S	Signed arithmetic modulo 2^8 or 2^{16} , sets CPSR GE bits
Q	Signed saturating arithmetic
SH	Signed arithmetic, halving results
U	Unsigned arithmetic modulo 2^8 or 2^{16} , sets CPSR GE bits
UQ	Unsigned saturating arithmetic
UH	Unsigned arithmetic, halving results

ARM Assembly

Load/Store

ARM Book

Table 3.4

ARM addressing modes

Syntax	Name
[Rn, #±<offset_12>]	Immediate offset
[Rn, ±Rm, <shift_op> #<shift>]	Scaled register offset
[Rn, #±<offset_12>]!	Immediate pre-indexed
[Rn, ±Rm, <shift_op> #<shift>]!	Scaled register pre-indexed
[Rn], #±<offset_12>	Immediate post-indexed
[Rn], ±Rm, <shift_op> #<shift>	Scaled register post-indexed

ARM Assembly

Load/Store

ARM Book

Register immediate: [Rn]

When using immediate offset mode with an offset of zero, the comma and offset can be omitted. That is, [Rn] is just shorthand notation for [Rn, #0]. This shorthand is referred to as *register immediate* mode. For example, the following line of code:

```
ldr r3, [r2]
```

ldr

Immediate offset: [Rn, #±< offset_12 >]

The immediate offset (which may be positive or negative) is added to the contents of Rn. The result is used as the address of the item to be loaded or stored. For example, the following line of code:

```
ldr r0, [r1, #12]
```


ARM Ref Manual

Using this book

This book is organized into the following chapters:

Part A Instruction Set Overview

Chapter A1 Overview of AArch32 state

Gives an overview of the AArch32 state.

Part B Advanced SIMD and Floating-point Programming

Chapter B1 Advanced SIMD Programming

Describes Advanced SIMD assembly language programming.

Chapter B2 Floating-point Programming

Describes floating-point assembly language programming.

Part C A32/T32 Instruction Set Reference

Chapter C1 Condition Codes

Describes condition codes and conditional execution of A32 and T32 code.

Chapter C2 A32 and T32 Instructions

Describes the A32 and T32 instructions supported in AArch32 state.

Chapter C3 Advanced SIMD Instructions (32-bit)

Describes Advanced SIMD assembly language instructions.

Chapter C4 Floating-point Instructions (32-bit)

Describes floating-point assembly language instructions.

Chapter C5 A32/T32 Cryptographic Algorithms

Lists the cryptographic algorithms that A32 and T32 SIMD instructions support.

ARM Assembly

LDR

ARM Ref

LDR (immediate offset)

Load with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

`LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset`

`LDR{type}{cond} Rt, [Rn, #offset]! ; pre-indexed`

`LDR{type}{cond} Rt, [Rn], #offset ; post-indexed`

`LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doubleword`

`LDRD{cond} Rt, Rt2, [Rn, #offset]! ; pre-indexed, doubleword`

`LDRD{cond} Rt, Rt2, [Rn], #offset ; post-indexed, doubleword`

where:

type

can be any one of:

- B** unsigned Byte (Zero extend to 32 bits on loads.)
- SB** signed Byte (LDR only. Sign extend to 32 bits.)
- H** unsigned Halfword (Zero extend to 32 bits on loads.)
- SH** signed Halfword (LDR only. Sign extend to 32 bits.)
- omitted, for Word.

cond

is an optional condition code.

Rt

is the register to load.

Rn

is the register on which the memory address is based.

Rm

is a register containing a value to be used as the offset. $-Rm$ is not permitted in T32 code.

shift

is an optional shift.

Rt2

is the additional register to load for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset register and shift options

The following table shows the ranges of offsets and availability of these instructions:

ARM Assembly

LDR

ARM Ref

LDR (register offset)

Load with register offset, pre-indexed register offset, or post-indexed register offset.

Syntax

```
LDR{type}{cond} Rt, [Rn, ±Rm {, shift}] ; register offset
LDR{type}{cond} Rt, [Rn, ±Rm {, shift}]! ; pre-indexed ; A32 only
LDR{type}{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed ; A32 only
LDRD{cond} Rt, Rt2, [Rn, ±Rm] ; register offset, doubleword ; A32 only
LDRD{cond} Rt, Rt2, [Rn, ±Rm]! ; pre-indexed, doubleword ; A32 only
LDRD{cond} Rt, Rt2, [Rn], ±Rm ; post-indexed, doubleword ; A32 only
```

where:

type

can be any one of:

B	unsigned Byte (Zero extend to 32 bits on loads.)
SB	signed Byte (LDR only. Sign extend to 32 bits.)
H	unsigned Halfword (Zero extend to 32 bits on loads.)
SH	signed Halfword (LDR only. Sign extend to 32 bits.)
-	omitted, for Word.

cond

is an optional condition code.

Rt

is the register to load.

Rn

is the register on which the memory address is based.

Rm

is a register containing a value to be used as the offset. -Rm is not permitted in T32 code.

shift

is an optional shift.

Rt2

is the additional register to load for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset register and shift options

The following table shows the ranges of offsets and availability of these instructions:

ARM Assembly

Load Immediate (li)

ARM Book

3.6.1 LOAD IMMEDIATE

3.6 Pseudo-Instructions

This pseudo-instruction loads a register with any 32-bit value:

ldr Load Immediate

When this pseudo-instruction is encountered, the assembler first determines whether or not it can substitute a `mov Rd, #<immediate>` or `mvn Rd, #<immediate>` instruction. If that is not possible, then it reserves four bytes in a “literal pool” and stores the immediate value there. Then, the pseudo-instruction is translated into an `ldr` instruction using Immediate Offset addressing mode with the `pc` as the base register.

Syntax

```
ldr{<cond>} Rd, =<immediate>
```

- The optional `<cond>` can be any of the codes from [Table 3.2](#) specifying conditional execution.
- The `<immediate>` parameter is any valid 32-bit quantity.

ARM Assembly

MOV

ARM Ref

C2.58 MOV

Move.

Syntax

`MOV{S}{cond} Rd, Operand2`

`MOV{cond} Rd, #imm16`

where:

S

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

cond

is an optional condition code.

Rd

is the destination register.

Operand2

is a flexible second operand.

imm16

is any value in the range 0-65535.

Operation

The MOV instruction copies the value of *Operand2* into *Rd*.

In certain circumstances, the assembler can substitute MVN for MOV, or MOV for MVN. Be aware of this when reading disassembly listings.

ARM Assembly

Syscall

ARM Ref

C2.145 SVC

SuperVisor Call.

Syntax

`SVC{cond} #imm`

where:

`cond`

is an optional condition code.

`imm`

is an expression evaluating to an integer in the range:

- 0 to $2^{24}-1$ (a 24-bit value) in an A32 instruction.
- 0-255 (an 8-bit value) in a T32 instruction.

Operation

The SVC instruction causes an exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SVC vector.

`imm` is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

Note

SVC was called `SWI` in earlier versions of the A32 assembly language. `SWI` instructions disassemble to SVC, with a comment to say that this was formerly `SWI`.

Condition flags

This instruction does not change the flags.

C2.112 SMC

Secure Monitor Call.

Syntax

`SMC{cond} #imm4`

where:

`cond`

is an optional condition code.

`imm4`

is a 4-bit immediate value. This is ignored by the Arm processor, but can be used by the SMC exception handler to determine what service is being requested.

SWI → SVC

ARM Conditionals

ARM Ref

C1.10 Condition code suffixes and related flags

Condition code suffixes define the conditions that must be met for the instruction to execute.

The following table shows the condition codes that you can use and the flag settings they depend on:

Table C1-1 Condition code suffixes

Suffix	Meaning
EQ	Equal
NE	Not equal
CS	Carry set (identical to HS)
HS	Unsigned higher or same (identical to CS)
CC	Carry clear (identical to LO)
LO	Unsigned lower (identical to CC)
MI	Minus or negative result
PL	Positive or zero result
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (this is the default)

Table C1-2 Condition code suffixes and related flags

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned \geq)
CC or LO	C clear	Lower (unsigned $<$)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned $>$)
LS	C clear or Z set	Lower or same (unsigned \leq)
GE	N and V the same	Signed \geq
LT	N and V differ	Signed $<$
GT	Z clear, N and V the same	Signed $>$
LE	Z set, N and V differ	Signed \leq
AL	Any	Always. This suffix is normally omitted.

ARM Conditionals

ARM Ref

The optional condition code is shown in syntax descriptions as {cond}. This condition is encoded in A32 instructions. For T32 instructions, the condition is encoded in a preceding IT instruction. An instruction with a condition code is only executed if the condition flags meet the specified condition.

The following is an example of conditional execution in A32 code:

```
ADD    r0, r1, r2    ; r0 = r1 + r2, don't update flags
ADDS   r0, r1, r2    ; r0 = r1 + r2, and update flags
ADDSCS r0, r1, r2    ; If C flag set then r0 = r1 + r2,
; and update flags
CMP    r0, r1        ; update flags based on r0-r1.
```

ADDS**CS**

gcd

```
CMP    r0, r1
SUBGT  r0, r0, r1
SUBLE  r1, r1, r0
BNE    gcd
```

SUB**GT**
SUB**LE**

CMP

In C the gcd algorithm can be expressed as:

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

The following examples show implementations of the gcd algorithm with and without conditional instructions.

Example of conditional execution using branches in A32 code

This example is an A32 code implementation of the gcd algorithm. It achieves conditional execution by using conditional branches, rather than individual conditional instructions:

```
gcd    CMP    r0, r1
        BEQ    end
        BLT    less
        SUBS   r0, r0, r1 ; could be SUB r0, r0, r1 for A32
        gcd
less   SUBS   r1, r1, r0 ; could be SUB r1, r1, r0 for A32
        B      gcd
end
```

The code is seven instructions long because of the number of branches. Every time a branch is taken, the processor must refill the pipeline and continue from the new location. The other instructions and non-executed branches use a single cycle each.

ARM Assembly

Operand2+*shift*

ARM Ref

C2.5 Syntax of Operand2 as a register with optional shift

When you use an Operand2 register in an instruction, you can optionally also specify a shift value.

Syntax

***Rn* {, *shift*}**

where:

Rn

is the register holding the data for the second operand.

shift

is an optional constant or register-controlled shift to be applied to *Rn*. It can be one of:

ASR #*n*

arithmetic shift right *n* bits, $1 \leq n \leq 32$.

LSL #*n*

logical shift left *n* bits, $1 \leq n \leq 31$.

LSR #*n*

logical shift right *n* bits, $1 \leq n \leq 32$.

ROR #*n*

rotate right *n* bits, $1 \leq n \leq 31$.

RRX

rotate right one bit, with extend.

type *Rs*

register-controlled shift is available in Arm code only, where:

type

is one of ASR, LSL, LSR, ROR.

Rs

is a register supplying the shift amount, and only the least significant byte is used.

if omitted, no shift occurs, equivalent to LSL #0.

ARM Assembly

Q

ARM Ref

C2.7 Saturating instructions

Some A32 and T32 instructions perform saturating arithmetic.

The saturating instructions are:

- QADD.
- QDADD.
- QSUB.
- QSDSUB.
- SSAT.
- USAT.

❖ Saturating ::= limit on overflow

Some of the parallel instructions are also saturating.

Saturating arithmetic

Saturation means that, for some value of 2^n that depends on the instruction:

- For a signed saturating operation, if the full result would be less than -2^n , the result returned is -2^n .
- For an unsigned saturating operation, if the full result would be negative, the result returned is zero.
- If the full result would be greater than 2^n-1 , the result returned is 2^n-1 .

When any of these occurs, it is called saturation. Some instructions set the Q flag when saturation occurs.

————— Note —————

Saturating instructions do not clear the Q flag when saturation does not occur. To clear the Q flag, use an MSR instruction.

ARM Assembly

Shift

ARM Ref

Arithmetic shift right (ASR)

Arithmetic shift right by n bits moves the left-hand $32-n$ bits of a register to the right by n places, into the right-hand $32-n$ bits of the result. It copies the original bit[31] of the register into the left-hand n bits of the result.

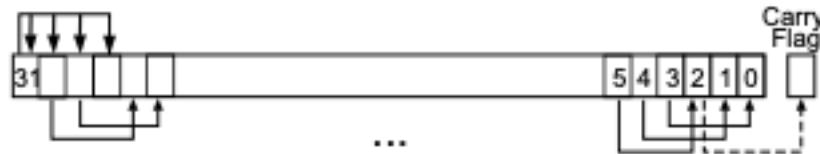


Figure C2-1 ASR #3

Logical shift right (LSR)

Logical shift right by n bits moves the left-hand $32-n$ bits of a register to the right by n places, into the right-hand $32-n$ bits of the result. It sets the left-hand n bits of the result to 0.



Figure C2-2 LSR #3

Logical shift left (LSL)

Logical shift left by n bits moves the right-hand $32-n$ bits of a register to the left by n places, into the left-hand $32-n$ bits of the result. It sets the right-hand n bits of the result to 0.

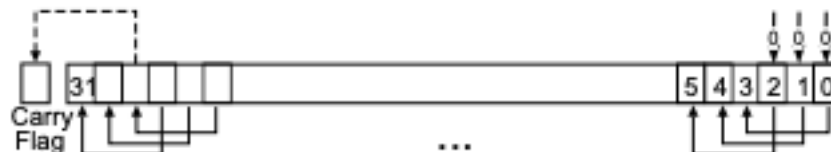


Figure C2-3 LSL #3

ARM Assembly

Rotate

ARM Ref

Rotate right (ROR)

Rotate right by n bits moves the left-hand $32-n$ bits of a register to the right by n places, into the right-hand $32-n$ bits of the result. It also moves the right-hand n bits of the register into the left-hand n bits of the result.



Figure C2-4 ROR #3

Rotate right with extend (RRX)

Rotate right with extend moves the bits of a register to the right by one bit. It copies the carry flag into bit[31] of the result.

When the instruction is `RRXS` or when `RRX` is used in *Operand2* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to bit[0] of the register *Rn*.



Figure C2-5 RRX

ARM Assembly

ARM Ref

A1.3 Processor modes, and privileged and unprivileged software execution

The Arm architecture supports different levels of execution privilege. The privilege level depends on the processor mode.

————— Note —————

Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline do not support the same modes as other Arm architectures and profiles. Some of the processor modes listed here do not apply to these architectures.

Table A1-1 AArch32 processor modes

Processor mode	Mode number
User	0b10000
FIQ	0b10001
IRQ	0b10010
Supervisor	0b10011
Monitor	0b10110
Abort	0b10111
Hyp	0b11010
Undefined	0b11011
System	0b11111

ARM Assembly

ARM Ref

Application
level view

System level view

	User	System	Hyp [†]	Supervisor	Abort	Undefined	Monitor [‡]	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

‡ Exists only in Secure state.

† Exists only in Non-secure state.

Cells with no entry indicate that the User mode register is used.

ARM Assembly

ARM Ref

dep·re·cate | 'deprə,kāt |

verb *[with object]*

1 express disapproval of: *what I deprecate is persistent indulgence.*

- **(be deprecated)** (chiefly of a software feature) be usable but regarded as obsolete and best avoided, typically due to having been superseded: *this feature is deprecated and will be removed in later versions* | **(as adjective deprecated)** : *avoid the deprecated <blink> element that causes text to flash on and off.*

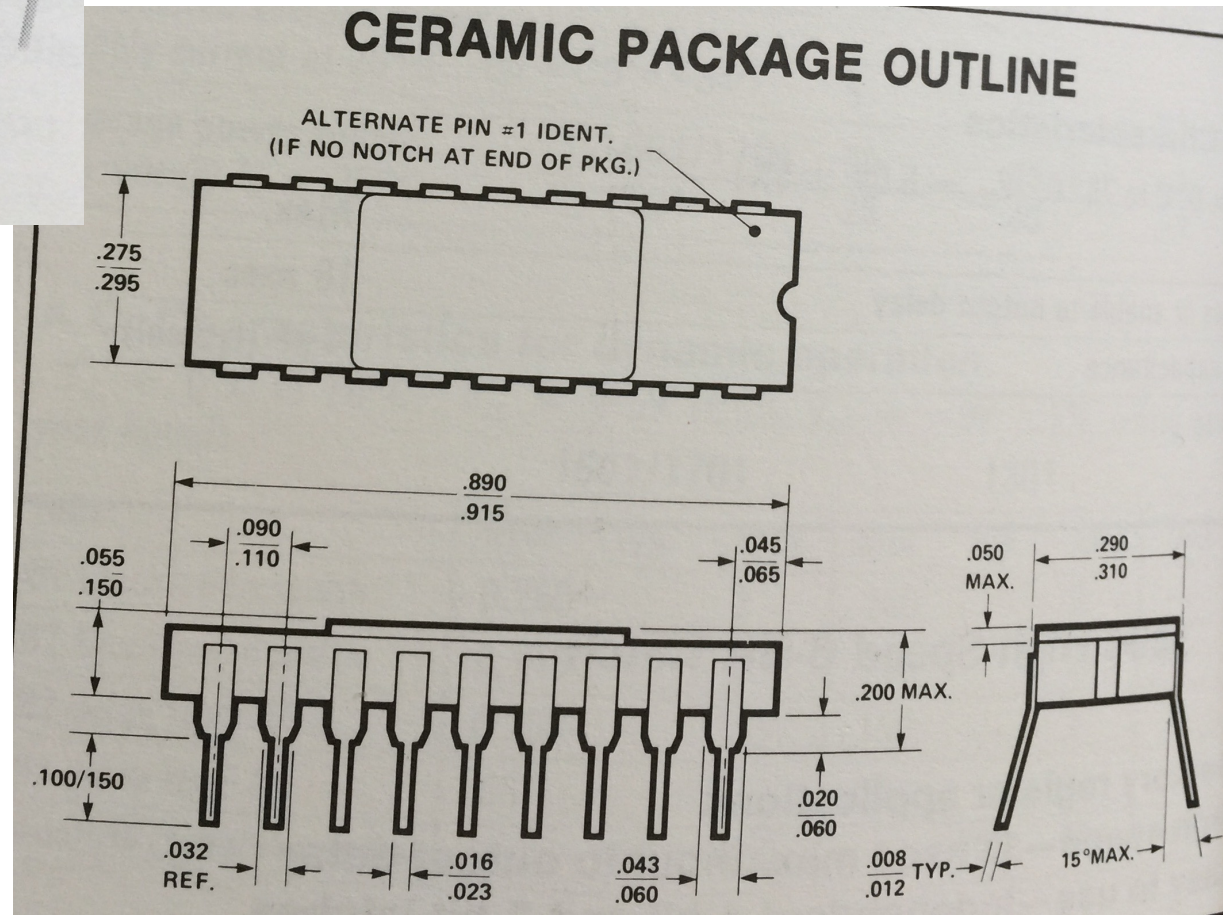
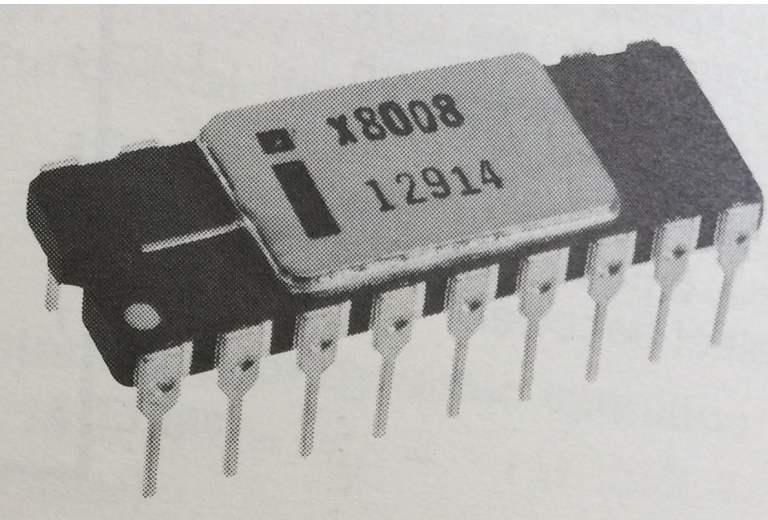
RISC vs CISC

Old **CISC**
8-Bit MPU's
(i8080)

➤ See separate slide set "MCS8"

i8008 MPU

MCS-8



i8080 Code



John Stephenson, Analyst programmer
magnetic tape reels.

Answered 9h ago

Set low byte A = all 1's or all 0's

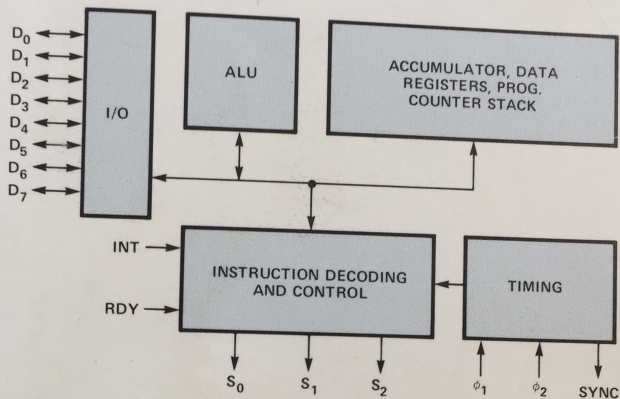
Several methods:

1	MOV	AL, FF	←	MOV
2				
3				
4	OR	AL, FF		
5				
6				
7	XOR	AL, AL		
8	NOT	AL		

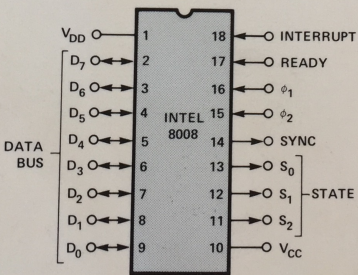
MCS-8: i8008

MCS-8

MCS-8TM MICRO COMPUTER SET



8008 BLOCK DIAGRAM



8008 PIN CONFIGURATION



3065 Bowers Ave., Santa Clara, Ca., 95051
Phone: (408) 246-7501

i8008 Block Diagram

Data bus

MCS-8

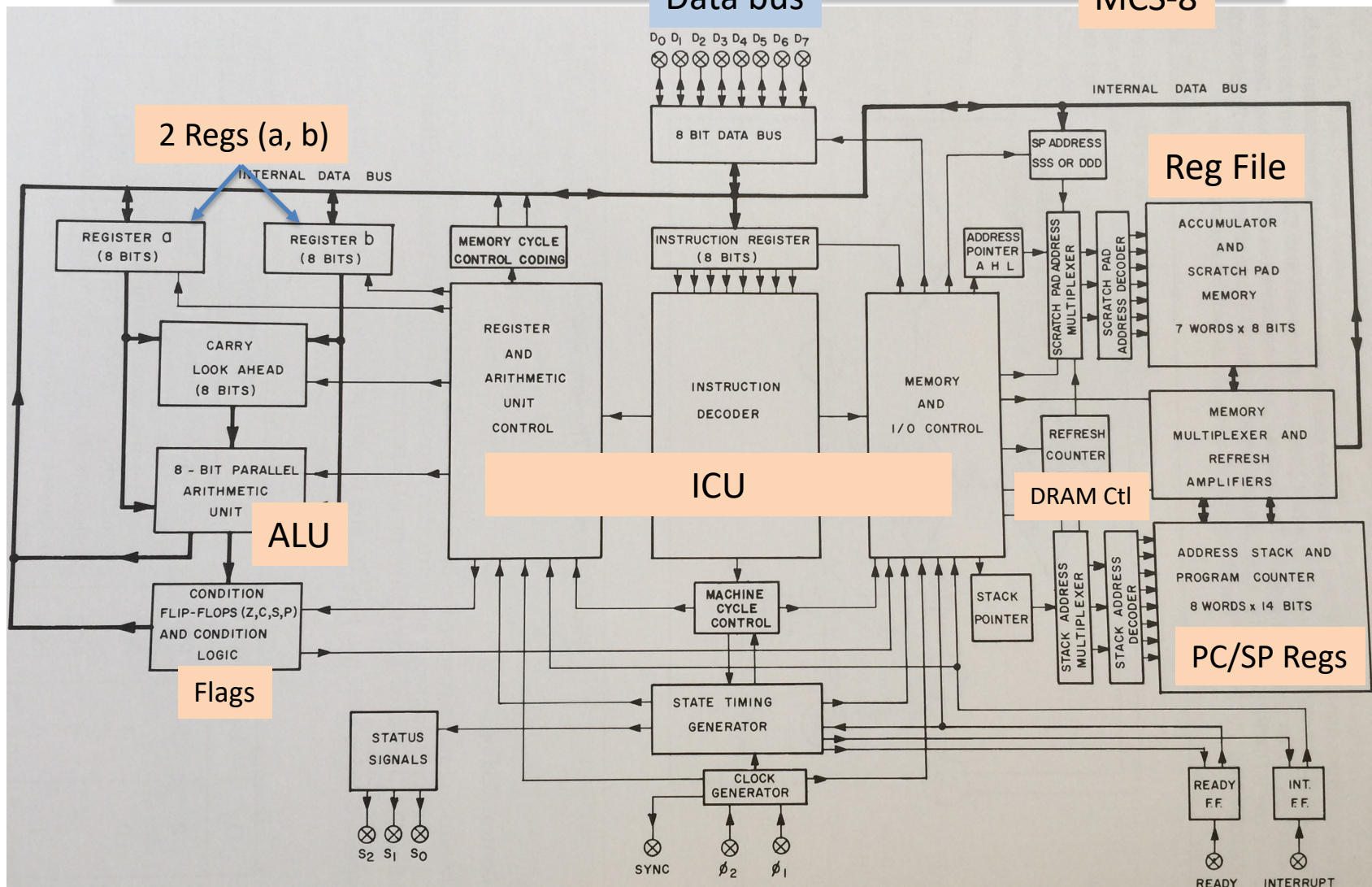


Figure 3. 8008 Block Diagram

System Block Diagram

MCS-8

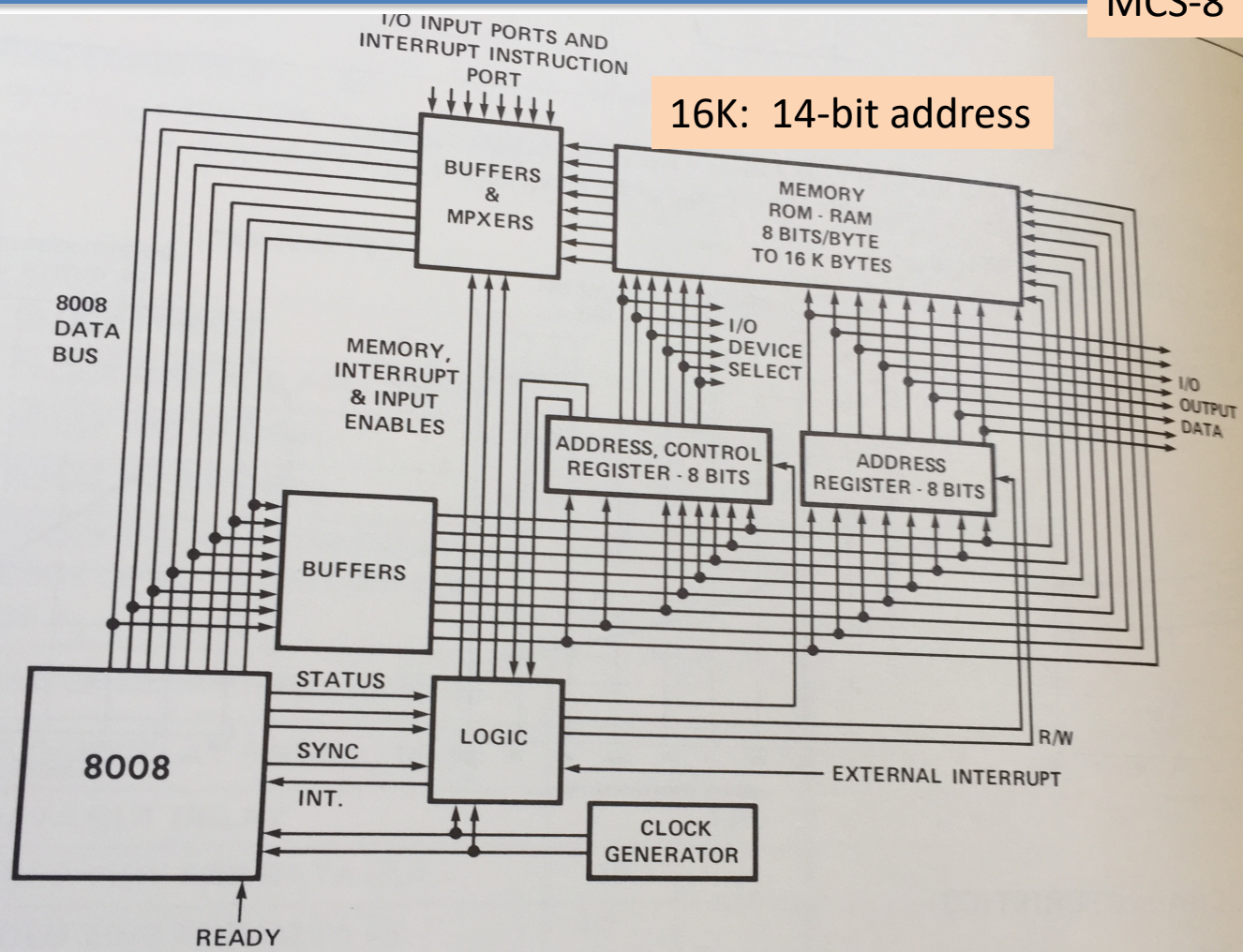
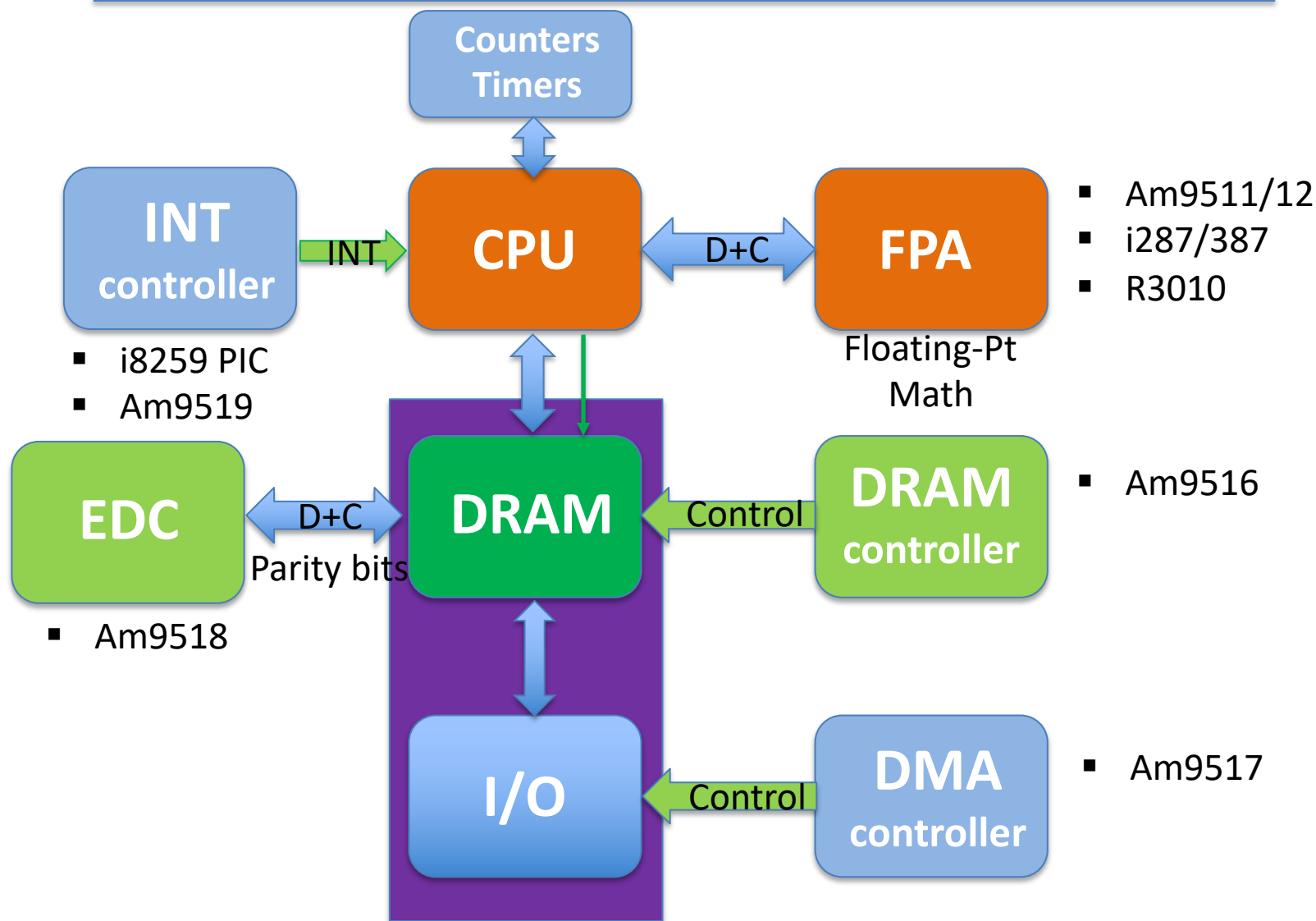


Figure 8. MCS-8 Basic System

CPU Peripherals



Computer Architecture

Interrupts

Interrupts vs Polling

Why do we use interrupts rather than polling in embedded systems?



Jeff Drobman

Lecturer at California State University, Northridge (2016–present) · Just now

we use either or both in embedded systems. it is a hardware vs software tradeoff, as interrupts need extra hardware logic. it will be cheaper to use polling for simple applications.

MIPS Interrupts

➤ NMI → Non *Maskable* ← Power-on

➤ NVI → Non *Vectored* ← BIOS

➤ VI → *Vectored*

▪ Vectors ← Devices:

Device ID
Placed on Data Bus

Read with **Load Byte**

- 1) Keyboard
- 2) Mouse
- 3) Display
- 4) Printer
- 5) USB

Interrupts

General

MIPS

CLASSES

❖ MASKABLE

☐ **NMI** (non-maskable)

- Power-ON Reset

☐ INT (maskable)

INT's (8)

◆ INT 0 (Pin 33)

◆ INT 1 (Pin 34)

◆ INT 2 (Pin 35)

◆ INT 7

❖ VECTORED

☐ **NVI** (non-V)

☐ **VI**

❖ PRIORITY (PIC)

☐ High

☐ Low (High INTs “preempt” Low)

❖ INTERNAL

☐ Hardware events

- **Timers**

- ADC

- I/O (S, P)

☐ Software exceptions

ENABLES

❖ GIE (2) – global (2 groups)

❖ *Mask*: INT 0-7

PRIORITIES

❖ HIGH

❖ LOW

→ SAVED ON STACK

❖ **PC**

❖ **STATUS**

❖ **CAUSE**

PROCESSOR STATE

Lab 4 INT Model

INT's Used: 4

❖ MASKABLE (3)

❑ 1 NMI (non-maskable)

▪ Power-ON Reset

❑ 2 INT (maskable)



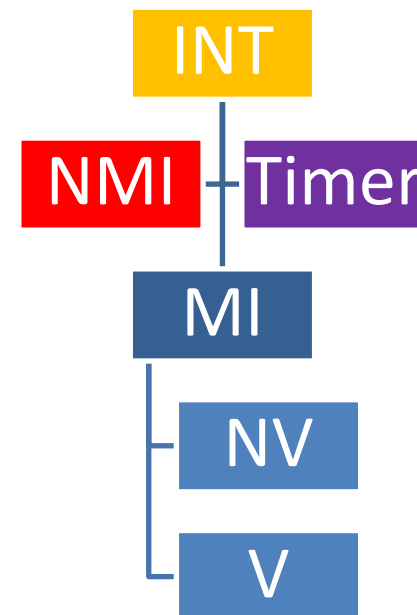
❖ VECTORED (2)

❑ 1 NVI (non)

❑ 1 VI

❖ TIMER (1)

Hierarchy: **Priority**



Interrupt Handlers

Lab 4

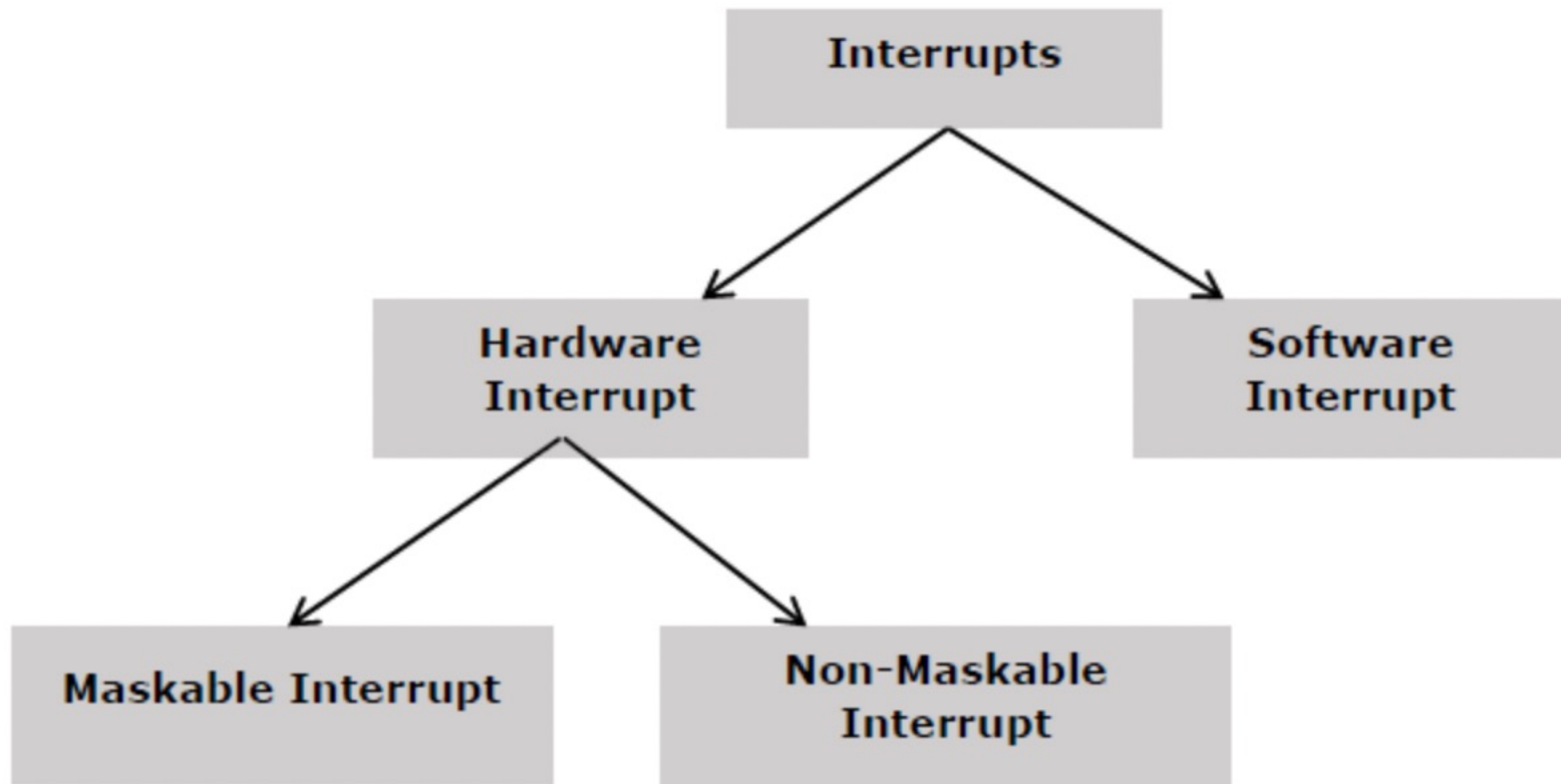
- ❖ Decode *Pending* Interrupts
- ❖ Allocate memory for Handlers
- ❖ Use *Jump Table*
 - ☐ Order by *Priority*
 - ☐ Test & Jump
 - ☐ Handlers as subroutines: `jal` → `jr $ra`

x86 Interrupts

COMP122 tutorialspoint.com

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an **ISR** (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.

The following image shows the types of interrupts we have in a 8086 microprocessor –



x86 Interrupts

Software Interrupts

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes –

INT- Interrupt instruction with type number

It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

Its execution includes the following steps –

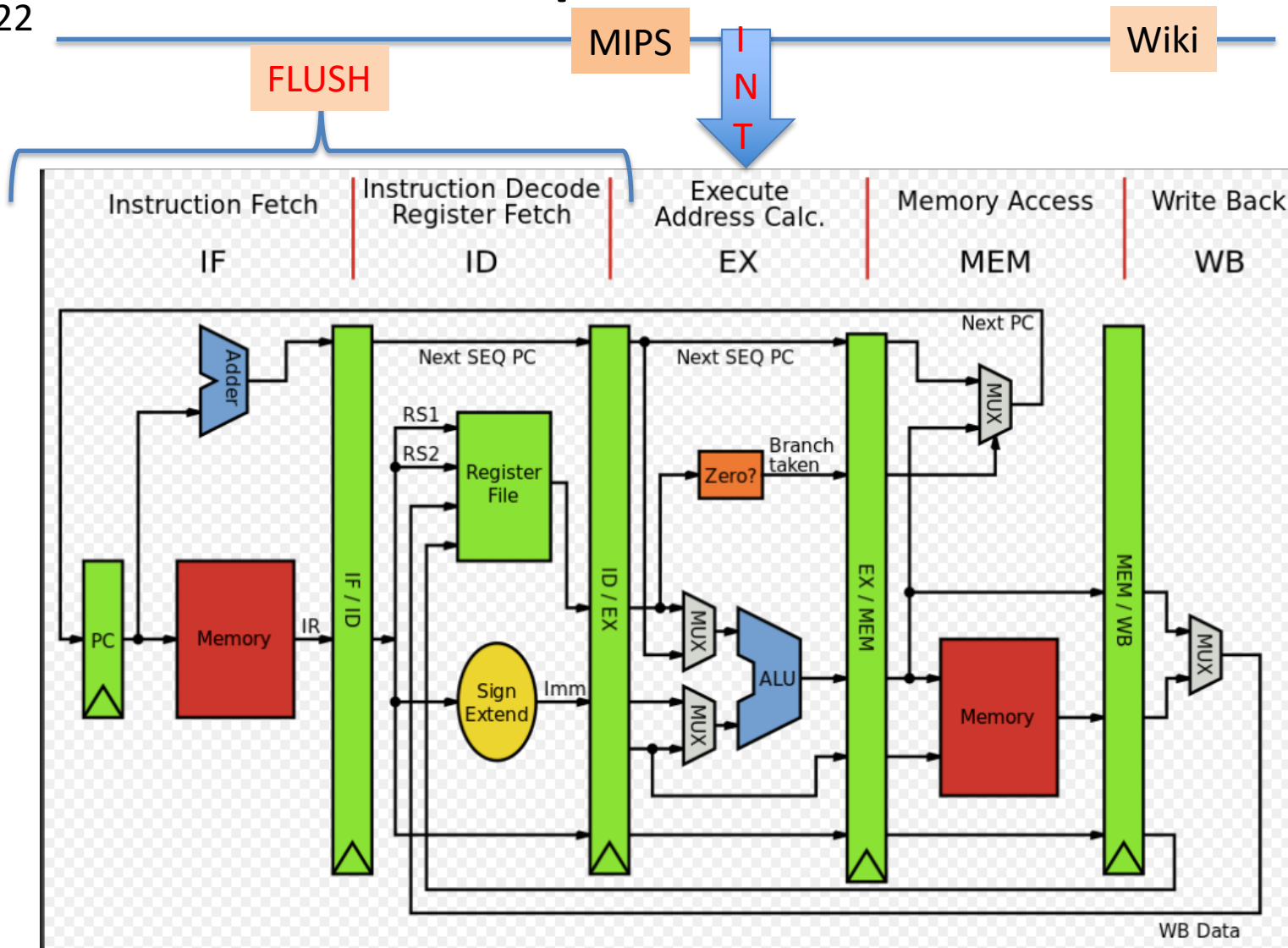
- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 'type number' \times 4
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

x86 SW Interrupts

The starting address for type0 interrupt is 000000H, for type1 interrupt is 00004H similarly for type2 is 00008H andso on. The first five pointers are dedicated interrupt pointers. i.e. –

- **TYPE 0** interrupt represents division by zero situation.
- **TYPE 1** interrupt represents single-step execution during the debugging of a program.
- **TYPE 2** interrupt represents non-maskable NMI interrupt.
- **TYPE 3** interrupt represents break-point interrupt.
- **TYPE 4** interrupt represents overflow interrupt.

MIPS Pipeline on INT



MIPS, showing the five stages (instruction fetch, instruction decode, execute, memory access and write back).

Interrupts & Exceptions

P&H Ch 7

COMP 122: Computer
Architecture and
Assembly Language
Spring 2020

7.7 Exceptions and interrupts

(Original section¹)

COD Section 4.9 (Exceptions) describes the MIPS exception facility, which responds both to exceptions caused by errors during an instruction's execution and to external interrupts caused by I/O devices. This section describes exception and *interrupt handling* in more detail.² In MIPS processors, a part of the CPU called coprocessor 0 records the information the software needs to handle exceptions and interrupts. The MIPS simulator SPIM does not implement all of coprocessor 0's registers, since many are not useful in a simulator or are part of the memory system, which SPIM does not implement. However, SPIM does provide the following coprocessor 0 registers:

Figure 7.7.1: Coprocessor 0 registers.

Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

[Feedback?](#)

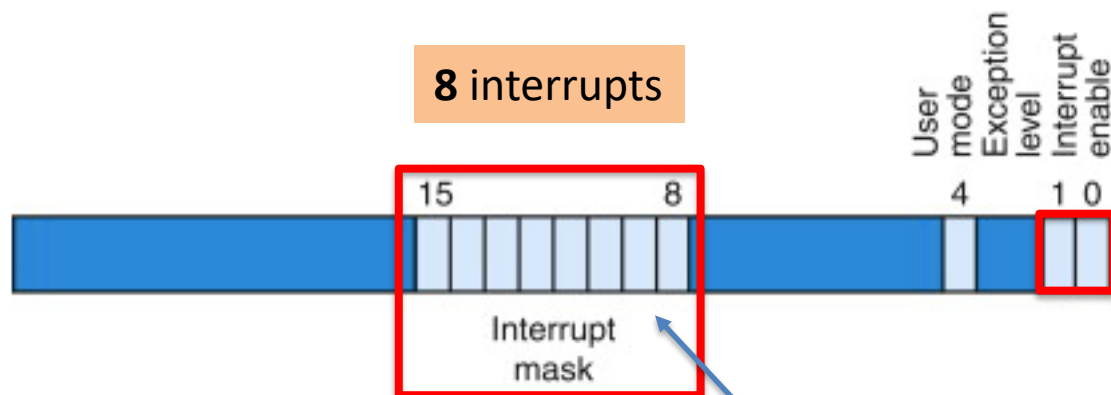
Interrupt handler. A piece of code that is run as a result of an exception or an interrupt.

Interrupts & Exceptions

P&H Ch 7

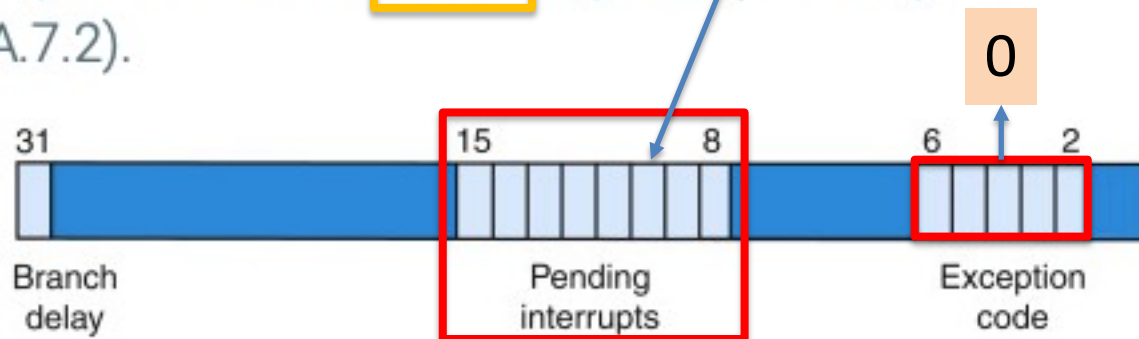
COMP 122: Computer
Architecture and
Assembly Language
Spring 2020

Figure 7.7.2: The **status** register (COD Figure A.7.1).



Mask(n) & Pending(n) → INT(n)

Figure 7.7.3: The **cause** register (COD Figure A.7.2).



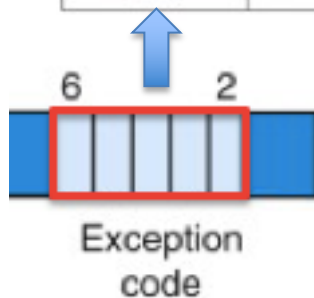
Interrupts & Exceptions

P&H Ch 7

COMP 122: Computer
Architecture and
Assembly Language
Spring 2020

Figure 7.7.4: Causes of exceptions.

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point



DEC PDP-11

1st LSI-chip Computer

1970

Wiki



PDP-11/40. The processor is at the bottom. A TU56 dual DECTape drive is installed above it.

DEC PDP-11

1970

Wiki



No dedicated I/O instructions [edit]

Early models of the PDP-11 had no dedicated [bus](#) for [input/output](#), but only a [system bus](#) called the [Unibus](#), as input and output devices were mapped to memory addresses.

An input/output device determined the memory addresses to which it would respond, and specified its own [interrupt vector](#) and [interrupt priority](#). This flexible

Interrupts [edit]

The PDP-11 supports hardware [interrupts](#) at [four priority levels](#). Interrupts are serviced by software service routines, which could specify whether they themselves [could be interrupted](#) (achieving [interrupt nesting](#)). The event that causes the interrupt is indicated by the device itself, as it informs the processor of the address of its own interrupt vector.

[Interrupt vectors](#) are blocks of two [16-bit words in low kernel address space](#) (which normally corresponded to low physical memory) between 0 and 776. The first word of the interrupt vector contains the [address of the interrupt service routine](#) and the second word the [value to be loaded into the PSW](#) (priority level) on entry to the service routine.

Instruction set orthogonality [edit]

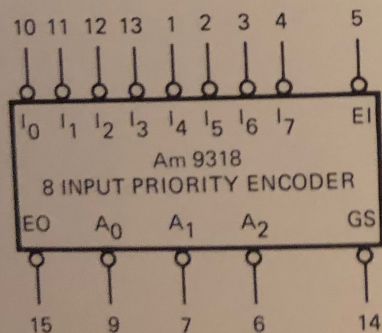
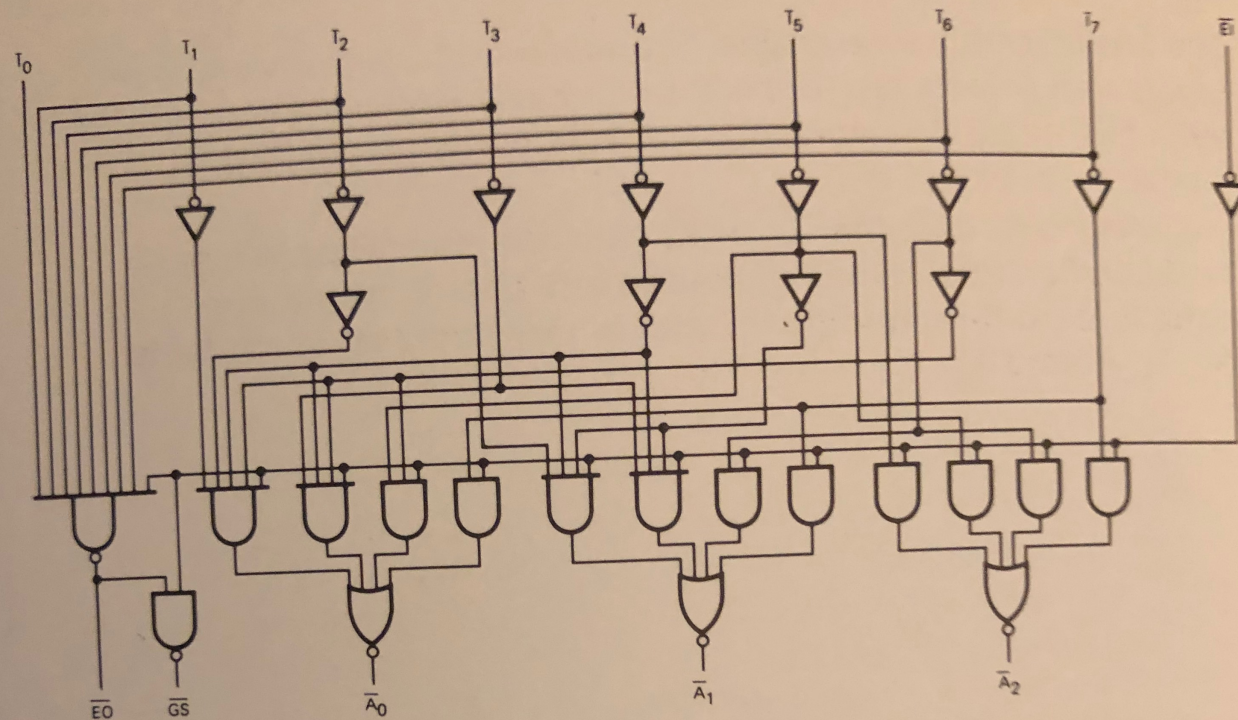
See also: [PDP-11 architecture](#)

The PDP-11 processor architecture has a mostly [orthogonal instruction set](#). For example, instead of instructions such as *load* and *store*, the PDP-11 has a *move* instruction for which either operand (source and destination) can be memory or register. There are no specific *input* or *output* instructions; the PDP-11 uses [memory-mapped I/O](#) and so the same *move* instruction is used; orthogonality even enables moving data directly from an input device to an output device. More complex instructions such as *add* likewise can have memory, register, input, or output as source or destination.

Most operands can apply any of eight addressing modes to eight registers. The addressing modes provide register, immediate, absolute, relative, deferred (indirect), and indexed addressing, and can specify autoincrementation and autodecrementation of a register by one (byte instructions) or two (word instructions). Use of relative addressing lets a machine-language program be [position-independent](#).

PIC: Priority Interrupt Enc

Logic Diagram/Symbol



Characteristics

Typical Delay \bar{I} to \bar{A} 16 ns

Typical Power Dissipation 250 mW

V_{CC} = Pin 16
GND = Pin 8

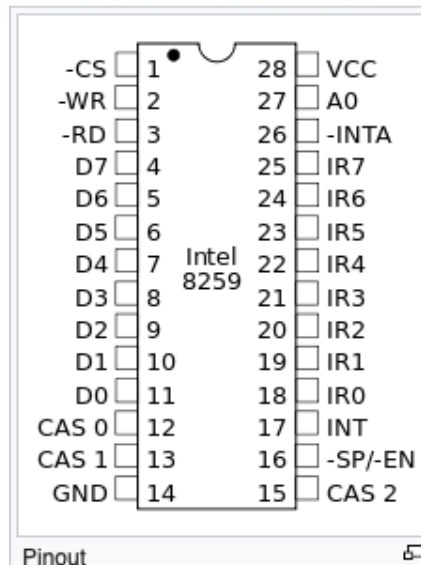
i8259 PIC

The **Intel 8259** is a **Programmable Interrupt Controller (PIC)** designed for the **Intel 8085** and **Intel 8086** microprocessors. The initial part was 8259, a later A suffix version was upward compatible and usable with the 8086 or **8088** processor. The 8259 combines multiple interrupt input sources into a single **interrupt** output to the host microprocessor, extending the interrupt levels available in a system beyond the one or two levels found on the processor chip. The 8259A was the interrupt controller for the **ISA bus** in the original **IBM PC** and **IBM PC AT**.

The 8259 was introduced as part of Intel's **MCS 85** family in 1976. The 8259A was included in the original PC introduced in 1981 and maintained by the **PC/XT** when introduced in 1983. A second 8259A was added with the introduction of the **PC/AT**. The 8259 has coexisted with the **Intel APIC Architecture** since its introduction in **Symmetric Multi-Processor PCs**. Modern PCs have begun to phase out the 8259A in favor of the **Intel APIC Architecture**. However, while not anymore a separate chip, the 8259A interface is still provided by the **Platform Controller Hub** or **Southbridge** chipset on modern **x86** motherboards.



Closeup of an Intel 8259A IRQ chip from a PC XT.



- ❖ IR0-7
- ❖ D0-7
- ❖ CAS0-3
- ❖ INTReq

i8259 PIC

- Trigger: Edge vs. Level
- Priority: Fixed vs. Rotating



- ❖ IRQ-7
- ❖ D0-7
- ❖ CAS0-3
- ❖ INTReq

Functional description [\[edit \]](#)

The main signal pins on an 8259 are as follows: eight interrupt input request lines named IRQ0 through IRQ7, an [interrupt request](#) output line named INTR, interrupt acknowledgment line named INTA, D0 through D7 for communicating the interrupt level or vector offset. Other connections include CAS0 through CAS2 for cascading between 8259s.

Up to eight *slave* 8259s may be cascaded to a *master* 8259 to provide up to 64 IRQs. 8259s are cascaded by connecting the INT line of one *slave* 8259 to the IRQ line of one *master* 8259.

There are three registers, an [Interrupt Mask Register](#) (IMR), an [Interrupt Request Register](#) (IRR), and an [In-Service Register](#) (ISR). The IRR maintains a [mask](#) of the current interrupts that are pending acknowledgement, the ISR maintains a mask of the interrupts that are pending an EOI, and the IMR maintains a mask of interrupts that should not be sent an acknowledgement.

[End Of Interrupt](#) (EOI) operations support specific EOI, non-specific EOI, and auto-EOI. A specific EOI specifies the IRQ level it is acknowledging in the ISR. A non-specific EOI resets the IRQ level in the ISR. Auto-EOI resets the IRQ level in the ISR immediately after the interrupt is acknowledged.

Edge and level interrupt trigger modes are supported by the 8259A. Fixed priority and rotating priority modes are supported.

The 8259 may be configured to work with an 8080/8085 or an 8086/8088. On the 8086/8088, the interrupt controller will provide an interrupt number on the data bus when an interrupt occurs. The interrupt cycle of the 8080/8085 will issue three bytes on the data bus (corresponding to a CALL instruction in the 8080/8085 instruction set).

The 8259A provides additional functionality compared to the 8259 (in particular buffered mode and level-triggered mode) and is upward compatible with it.

i8259 PIC

COMP122

x86 IRQs [[edit](#)]

Typically, on systems using the [Intel 8259](#) PIC, 16 IRQs are used. IRQs 0 to 7 are managed by one Intel 8259 PIC, and IRQs 8 to 15 by a second Intel 8259 PIC. The first PIC, the master, is the only one that directly signals the CPU. The second PIC, the slave, instead signals to the master on its IRQ 2 line, and the master passes the signal on to the CPU. There are therefore only 15 interrupt request lines available for hardware.

On newer systems using the [Intel APIC Architecture](#), typically there are 24 IRQs available, and the extra 8 IRQs are used to route PCI interrupts, avoiding conflict between dynamically configured PCI interrupts and statically configured ISA interrupts. On early APIC systems with only 16 IRQs or with only [Intel 8259](#) interrupt controllers, PCI interrupt lines were routed to the 16 IRQs using a PIR integrated into the southbridge.

The easiest way of viewing this information on [Windows](#) is to use [Device Manager](#) or [System Information](#) (msinfo32.exe). On [Linux](#), IRQ mappings can be viewed by executing `cat /proc/interrupts` or using the `procinfo` utility.

Master PIC [[edit](#)]

- IRQ 0 – [system timer](#) (cannot be changed)
- IRQ 1 – [keyboard controller](#) (cannot be changed)
- IRQ 2 – cascaded signals from IRQs 8–15 (any devices configured to use IRQ 2 will actually be using IRQ 9)
- IRQ 3 – [serial port controller](#) for [serial port 2](#) (shared with serial port 4, if present)
- IRQ 4 – serial port controller for serial port 1 (shared with serial port 3, if present)
- IRQ 5 – [parallel port 2 and 3](#) or [sound card](#)
- IRQ 6 – [floppy disk controller](#)
- IRQ 7 – parallel port 1. It is used for printers or for any parallel port if a printer is not present. It can also be potentially be shared with a secondary sound card with careful management of the port.

Slave PIC [[edit](#)]

- IRQ 8 – [real-time clock](#) (RTC)
- IRQ 9 – [Advanced Configuration and Power Interface](#) (ACPI) system control interrupt on Intel chipsets.^[2] Other chipset manufacturers might use another interrupt for this purpose, or make it available for the use of peripherals (any devices configured to use IRQ 2 will actually be using IRQ 9)
- IRQ 10 – The Interrupt is left open for the use of peripherals (open interrupt/available, SCSI or [NIC](#))
- IRQ 11 – The Interrupt is left open for the use of peripherals (open interrupt/available, SCSI or [NIC](#))
- IRQ 12 – [mouse](#) on [PS/2 connector](#)
- IRQ 13 – CPU [co-processor](#) or integrated [floating point unit](#) or [inter-processor interrupt](#) (use depends on OS)
- IRQ 14 – primary [ATA](#) channel (ATA interface usually serves [hard disk drives](#) and [CD drives](#))
- IRQ 15 – secondary ATA channel

Interrupt Handlers

How do you handle interrupt management (algorithms, computer architecture, operating systems, memory management, process scheduling, programming)?



Jeff Drobman, Lecturer at California State University, Northridge (2016-present)

Answered just now

there must be interrupt handler code (a subroutine) to process each interrupt and its source. microprocessors all have a mechanism to recognize enabled and pending interrupts. for example, MIPS has 8 interrupts, with enables and pending bits in its Status and Cause registers. upon an interrupt recognition by the ICU, the CPU vectors to a fixed address in the kernel text segment. there, code first disables some or all interrupts; saves some state (a set of registers) on the stack; then parses the pending interrupts, and uses a priority branch table to vector to the specific interrupt handler. at the end of processing, a "return from interrupt" (ERET in MIPS) is executed.

Interrupts C Example

```
#include <p18f4321.h>
void  ISR (void); //declares ISR as sub after main
#pragma code Int=0x08
void Intasm( )
{
    _asm //use assembly code
    GOTO ISR
    _endasm
}
#pragma code //org main
Void main( )
{
    // do stuff here
    While(1) {  }
}
#pragma interrupt ISR
Void ISR (void) //interrupt svc routine
{ //do int stuff
}
```


Config Interrupts in C

COMP122

EXTERNAL INTS

PIC18F

```
Void ISR( ); //declare the "ISR" subroutine
```

```
#pragma code int_vectH = 0x08 //assign int "vector" for High Pri Int
```

SET ORGS

```
Void IntH( ) {
```

```
_asm //use assembly code here (no "GOTO" in C)
GOTO ISR
_endasm
```

Or use a "Call":
ISR()

```
#pragma code int_vectL = 0x18 //assign int "vector" for Low Pri Int
```

```
#pragma code //main starts here (after the Ints)
```

```
Void main ( )
```

```
{
```

```
ADCON1=0x0F; //config Port B as input for interrupts
```

```
INTCONbits.INT0IE = 1 //enable INT0
```

```
INTCONbits.INT1IE = 1 //enable INT1
```

SETUP

```
INTCONbits.INT0F = 0 //clear flag
```

```
INTCONbits.INT1F = 0 //clear flag
```

```
INTCONbits.INT1IP = 0 //set INT1 to Low priority
```

```
RCONbits.IPEN = 1 //enable all priority interrupts
```

```
INTCONbits.GIEH = 1 //enable Global High priority interrupts
```

```
INTCONbits.GIEL = 1 //enable Global Low priority interrupts
```

```
While(1); //wait for INT0 or INT1
```

```
}
```

PINS

- ◆ INT 0 → RB0
- ◆ INT 1 → RB1
- ◆ INT 2 → RB2

Interrupts & Exceptions

P&H Ch 7

COMP 122: Computer
Architecture and
Assembly Language
Spring 2020

`.ktext 0x80000180`

```
mov $k1, $at      # Save $at register
sw  $a0, save0    # Handler is not re-entrant and can't use
sw  $a1, save1    # stack to save $a0, $a1
                        # Don't need to save $k0/$k1
```

```
mfc0 $k0, $13      # Move Cause into $k0

srl  $a0, $k0, 2    # Extract ExcCode field
andi $a0, $a0, 0xf

bgtz $a0, done      # Branch if ExcCode is Int (0)

mov  $a0, $k0       # Move Cause into $a0
mfco $a1, $14       # Move EPC into $a1
jal  print_excp     # Print exception error message
```

Interrupts & Exceptions

P&H Ch 7

**COMP 122: Computer
Architecture and
Assembly Language**
Spring 2020

done:

```

mfc0    $k0, $14      # Bump EPC
addiu    $k0, $k0, 4   # Do not re-execute
                        # faulting instruction
mtc0     $k0, $14      # EPC

mtc0     $0, $13       # Clear Cause register

mfc0     $k0, $12      # Fix Status register
andi     $k0, 0xffffd  # Clear EXL bit
ori      $k0, 0x1       # Enable interrupts
mtc0     $k0, $12

lw       $a0, save0     # Restore registers
lw       $a1, save1
mov      $at, $k1
    
```

eret

Return to EPC

.kdata

save0:
save1:

.word 0
.word 0

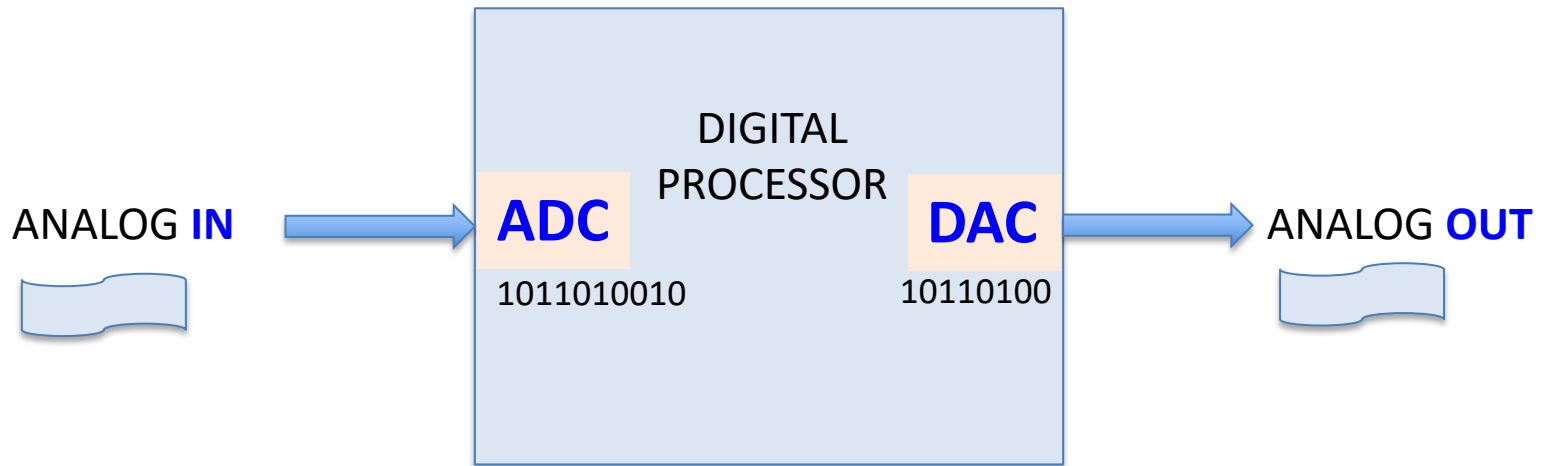
Section

Other Hardware

- ❖ ADC Converters
- ❖ Serial I/O

Data Conversion

Embedded Control lives in an ANALOG world



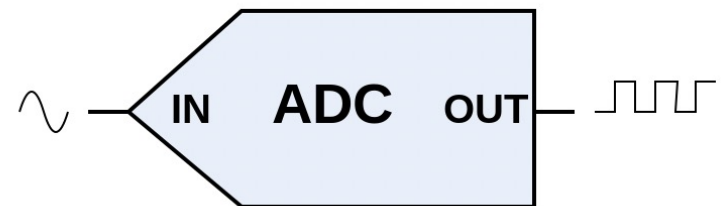
❖ ADC

- ✧ Typ 8-14 bits (resolution)
- ✧ Flash or SAR

❖ DAC

- ✧ Byte (8-bit)
- ✧ Resistor ladder

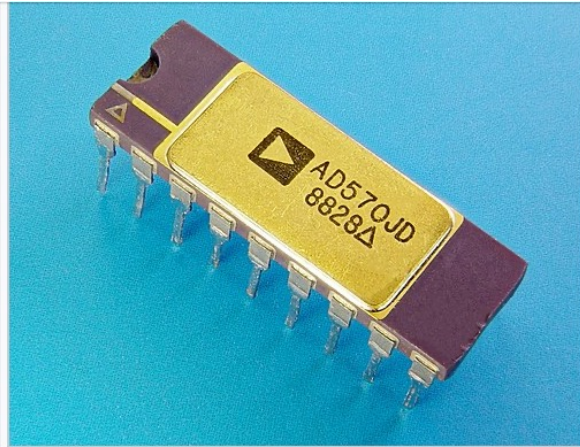
Electrical symbol [\[edit \]](#)



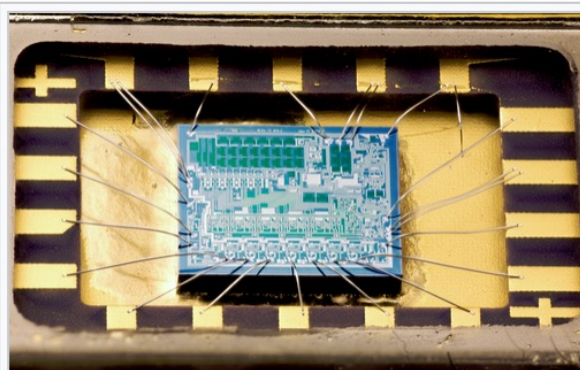
ADC Chips

Analog-to-digital converter

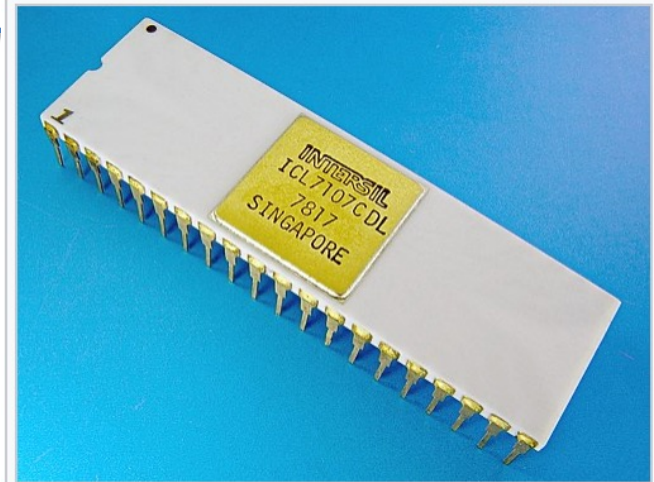
From Wikipedia, the free encyclopedia



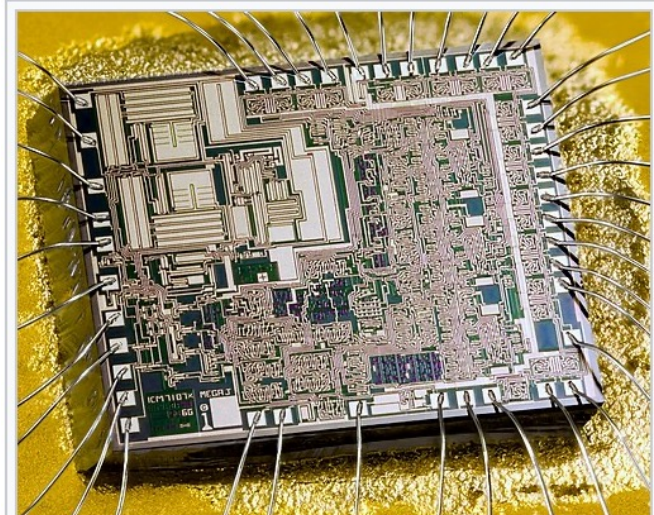
AD570 8-bit successive-approximation analog-to-digital converter



AD570/AD571 silicon die



INTERSIL ICL7107. 3 1/2 digit single-chip A/D converter



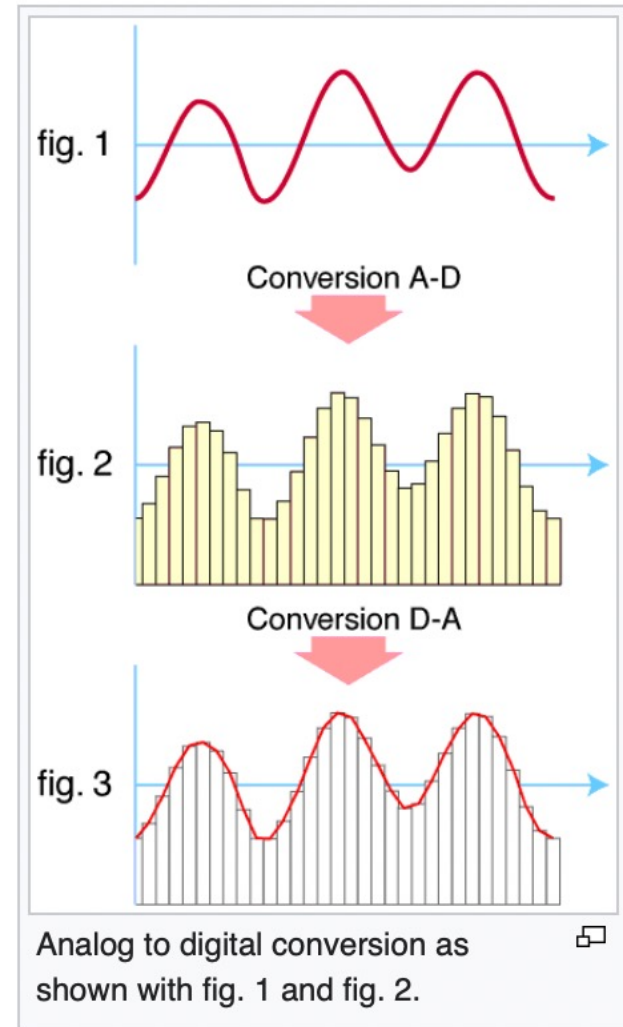
ICL7107 silicon die

Quantization

Analog-to-digital converter

From Wikipedia, the free encyclopedia

An ADC has several sources of errors. [Quantization](#) error and (assuming the ADC is intended to be linear) non-[linearity](#) are intrinsic to any analog-to-digital conversion. These errors are measured in a unit called the [least significant bit](#) (LSB). In the above example of an eight-bit ADC, an error of one LSB is $1/256$ of the full signal range, or about 0.4%.



SAR ADC

Analog-to-digital converter

From Wikipedia, the free encyclopedia

Successive approximation

A successive-approximation ADC uses a **comparator** and a **binary search** to successively narrow a range that contains the input voltage. At each successive **step**, the converter compares the input voltage to the output of an **internal digital to analog converter** which initially represents the midpoint of the allowed input voltage range. At each step in this process, the approximation is stored in a successive approximation register (**SAR**) and the output of the digital to analog converter is updated for a comparison over a narrower range.

Flash ADC

Analog-to-digital converter¹

From Wikipedia, the free encyclopedia

Direct-conversion

A **direct-conversion** or **flash** ADC has a bank of [comparators](#) sampling the input signal in parallel, each firing for a specific voltage range. The comparator bank feeds a digital [encoder logic circuit](#) that generates a binary number on the output lines for each voltage range.

ADCs of this type have a **large die size** and **high power** dissipation. They are often used for [video](#), [wideband communications](#), or other fast signals in [optical](#) and [magnetic storage](#).

The circuit consists of a **resistive divider network**, a set of **op-amp comparators** and a **priority encoder**. A small amount of **hysteresis** is built into the comparator to resolve any problems at voltage boundaries. At each node of the resistive divider, a comparison voltage is available. The purpose of the circuit is to compare the analog input voltage with each of the node voltages.

The circuit has the advantage of high speed as the conversion takes place **simultaneously** rather than sequentially. Typical conversion time is **100 ns** or less. Conversion time is limited only by the speed of the comparator and of the priority encoder. This type of ADC has the disadvantage that the **number of comparators required almost doubles for each added bit**. Also, the larger the value of n , the more complex is the priority encoder.

Maxim/ADI ADC



NOW PART OF



Part Number <input type="text" value="Filter by part number"/>	Resolution (bits) ↑↓	# Input Channels ↑↓	Sample Rate (max) (Msps) ↑↓	Data Bus Interface
MAX19191 Ultra-Low-Power, 10Msps, 8-Bit ADC	8	1	10	μP/8
MAX19507 Dual-Channel, 8- Bit, 130Msps ADC	8	2	130	Selectable Dual/Mux'd CMOS
MAX19506 Dual-Channel, 8- Bit, 100Msps ADC	8	2	100	Selectable Dual/Mux'd CMOS
MAX19505 Dual-Channel, 8- Bit, 65Msps ADC	8	2	65	Selectable Dual/Mux'd CMOS
MAX19515 Dual-Channel, 10-Bit, 65Msps ADC	10	2	65	Selectable Dual/Mux'd CMOS
MAX19517 Dual-Channel, 10-Bit, 130Msps ADC	10	2	130	Selectable Dual/Mux'd CMOS

Maxim/ADI ADC



Precision ADCs ($\leq 5\text{Msps}$)

Part Number \updownarrow <div>Filter by part number</div>	Resolution (ADC) (bits) \updownarrow	# Input Channels	Conv. Rate (max) (ksps) \updownarrow	Data Bus	ADC Architecture \updownarrow
MAX11410A 24-Bit Multi-Channel Low-Power 1.9ksps Delta-Sigma ADC with PGA	24	10	1.92	SPI	Sigma-Delta
MAX19777 3Msps, Ultra-tiny, Low Power, 2 Channel, Serial 12-Bit ADC	12	2	3000	SPI	SAR
MAX11261 24-Bit, 6-Channel, 16ksps, 6.2nV/ $\sqrt{\text{Hz}}$ PGA, Delta-Sigma ADC with I2C Interface	24	6	16	I ² C	Sigma-Delta

Peripherals

❖ Serial I/O

- ☐ USB
- ☐ I2C
- ☐ SIO
- ☐ UART (RS-232C)

❖ Counters/Timers

- ☐ GP
- ☐ Watchdog Timer

Serial I/O



Universal Asynchronous Receiver/Transmitter (UART)

UARTs

SPI

I2c

Microcontrollers

Embedded Systems

+2



Are UART, SPI, and I2C implementations microcontroller dependent?

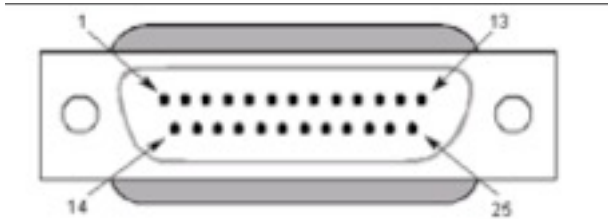


Jeff Drobman, works at Dr Jeff Software

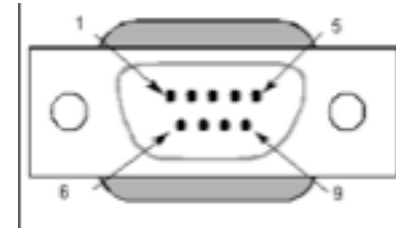
Answered just now

they are all industry standard serial communications. they include protocols implemented in software, plus hardware that includes serial-to-parallel conversion using registers. some of that hardware is included on some microcontrollers.

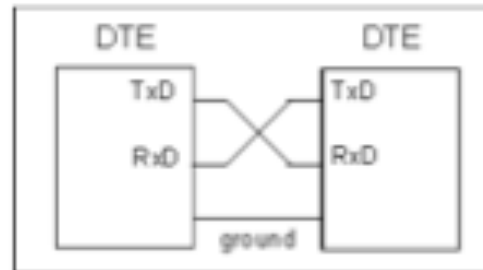
RS 232



DB-25



DB-9



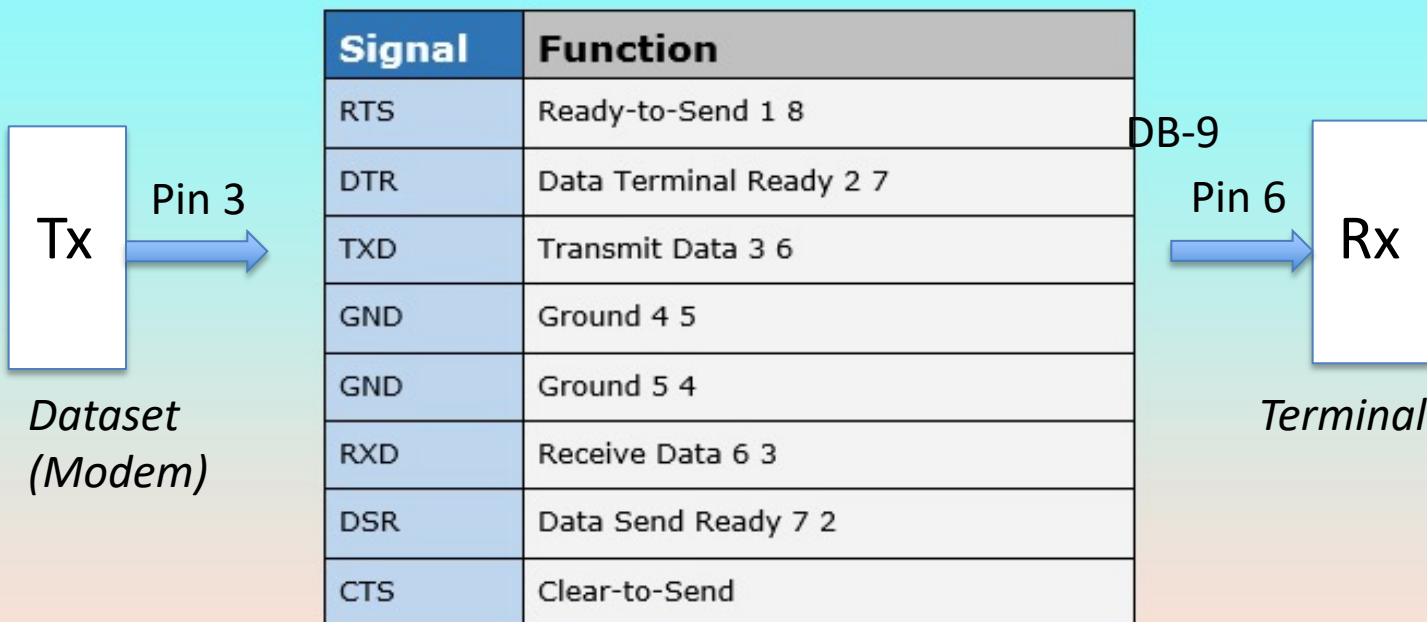
"Null Modem" cable

EIA (RS) 232

EIA232

Electronics Industries Association (EIA) 232 (RS232) is a serial communication standard, which you must know while connecting the PIC trainer. It is a standard for transmitting data serially. The RS232 protocol running on the console port is the same communications protocol format used on a computer's COM and COM2 ports. In PIC18F, the USART module supports RS232 operation using internal oscillator block. The connector for the serial communications port on the computer is either **DB-9** or **DB-25** type connector.

The table below shows the pin assignment for the RS232 associated with the DB-9 connector:



Networks

❖ Ethernet (IEEE 802.3)

- ☐ MAC (PMI + Layer 2)
- ☐ PHY (PMD: Layer 1)

❖ WiFi (IEEE 802.11)

- ☐ MAC (PMI + Layer 2)
- ☐ PHY (PMD: Layer 1)

Computer Architecture

ICU

software is executed by any computer according to the rules of its instruction set (ISA), and implemented by lots of logic in a state machine called the Instruction Control Unit, which produces a set of bits to control the entire computer each clock cycle.

- ❖ ***FSM vs Microprogramming***
- ❖ ***Pipelining***

Instruction Decode

ICU

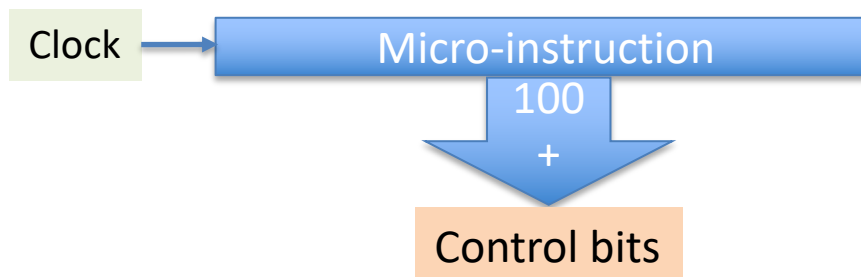
How does a computer read instructions? How does a computer know that "01" is to make a mov operation and that "10" is to make a xor operation?



Jeff Drobman, Lecturer at California State University, Northridge (2016-present)

Answered just now

a CPU has a large set of control bits (typ 100+) that are generated by the "Control Unit" logic by decoding the opcode and other instruction fields. the ALU operation itself has typ. 4 control bits to select 1/16 operations (add, sub, xor, etc.). there are also bits to select the inputs to each side of the ALU via its "mux".



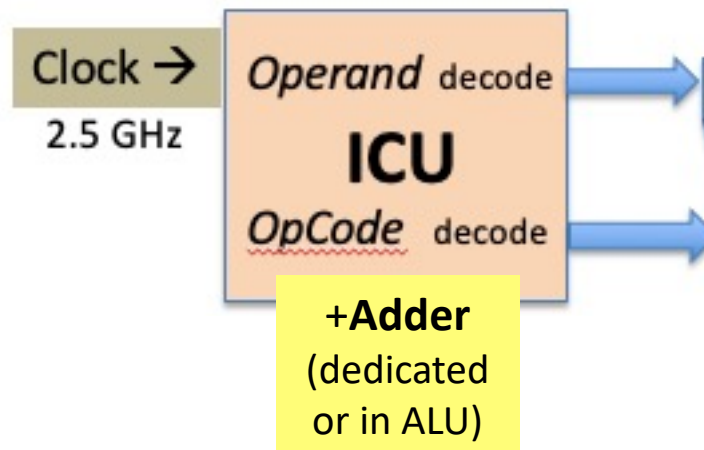
ICU

COMP122

❑ Instruction Control Unit (ICU)

❖ Decode

- OpCode
- Operands
 - ALU muxes
 - GR dest mux



❖ Calculate (adder)

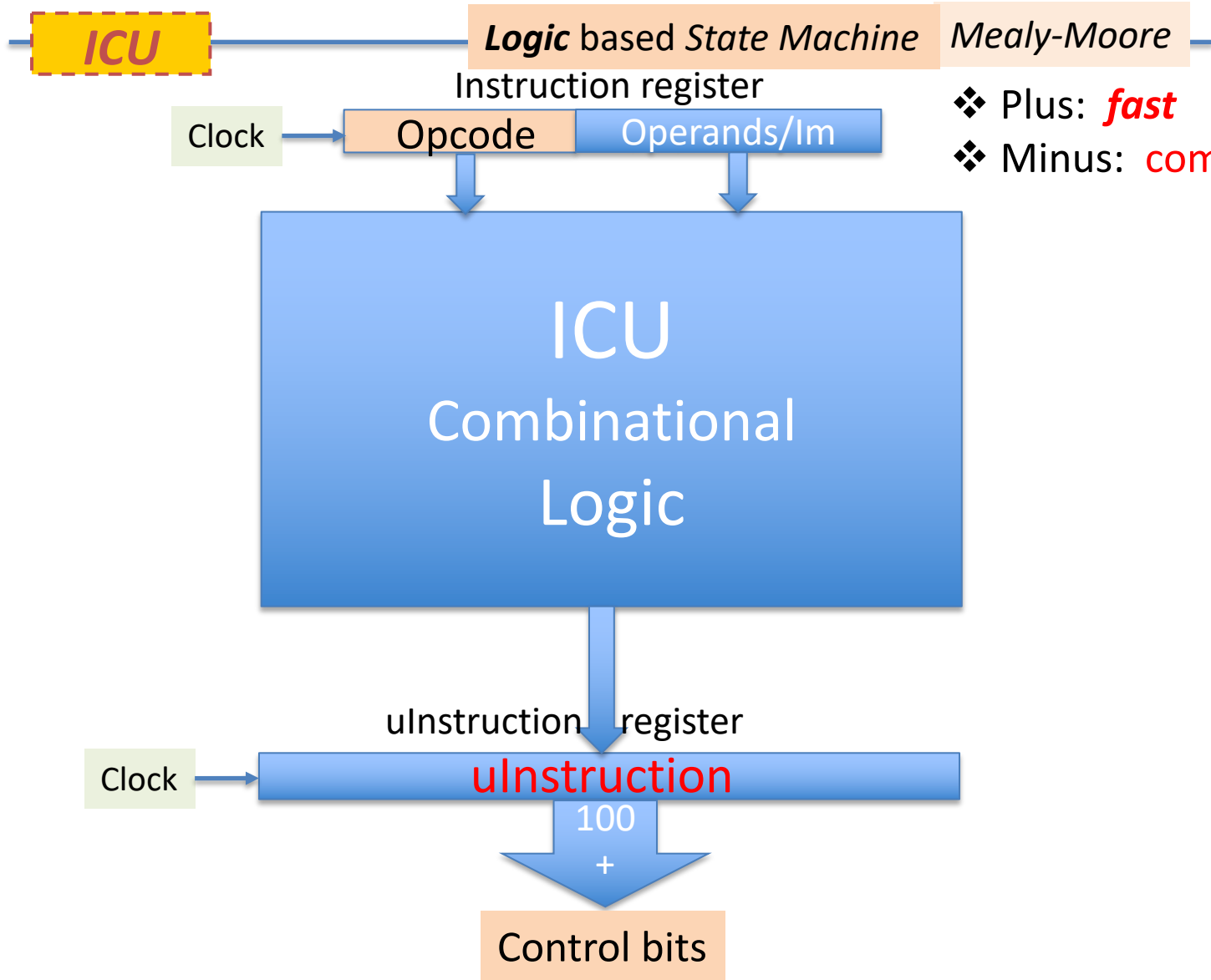
- *Effective Address* (EA)
 - Load/Store: GR + Immed/offset (indexed)
 - Branches: PC + offset

❖ *FSM vs Microprogramming*

❖ *Pipelining*

ICU: FSM

COMP122



- ❖ Plus: **fast**
- ❖ Minus: **complex** logic

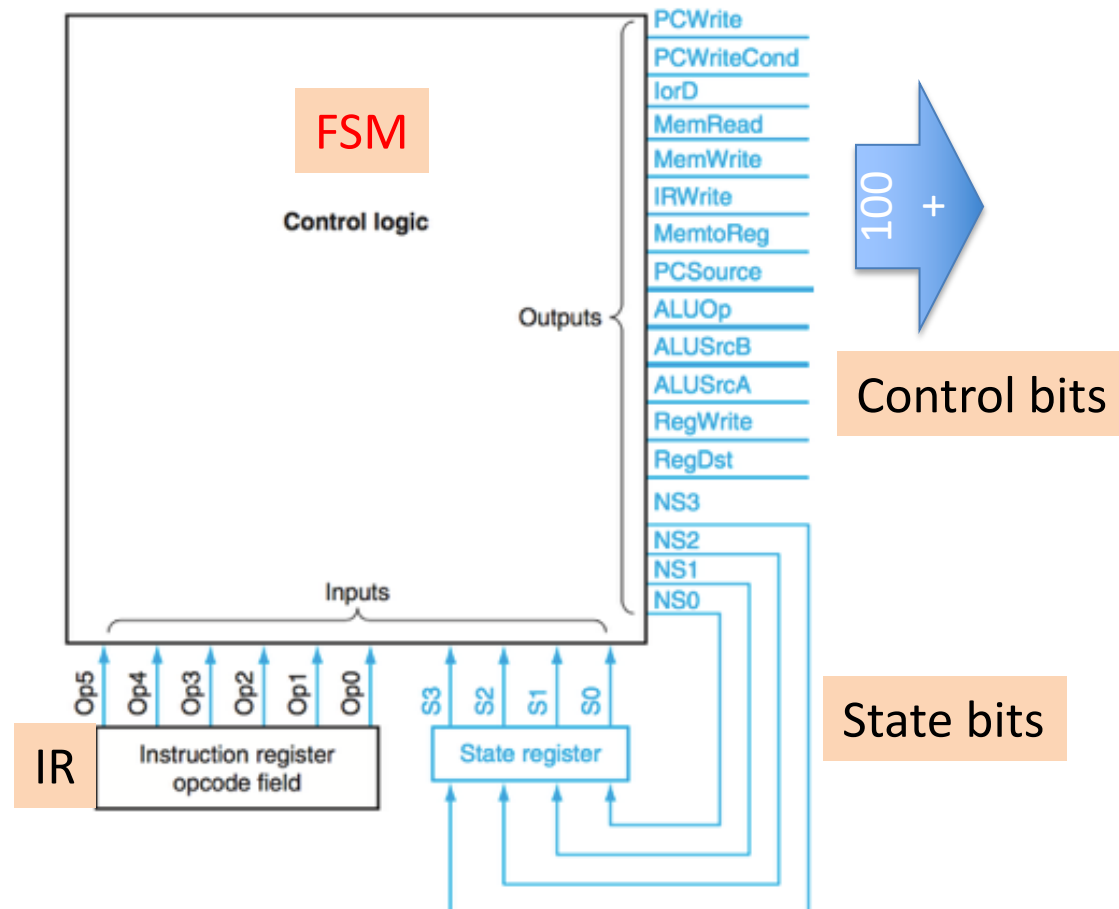
ICU: FSM

Hennessy & Patterson

Ch 10

Figure 10.3.2: The control unit for MIPS will consist of some control logic and a register to hold the state (COD Figure D.3.2).

The state register is written at the active clock edge and is stable during the clock cycle.

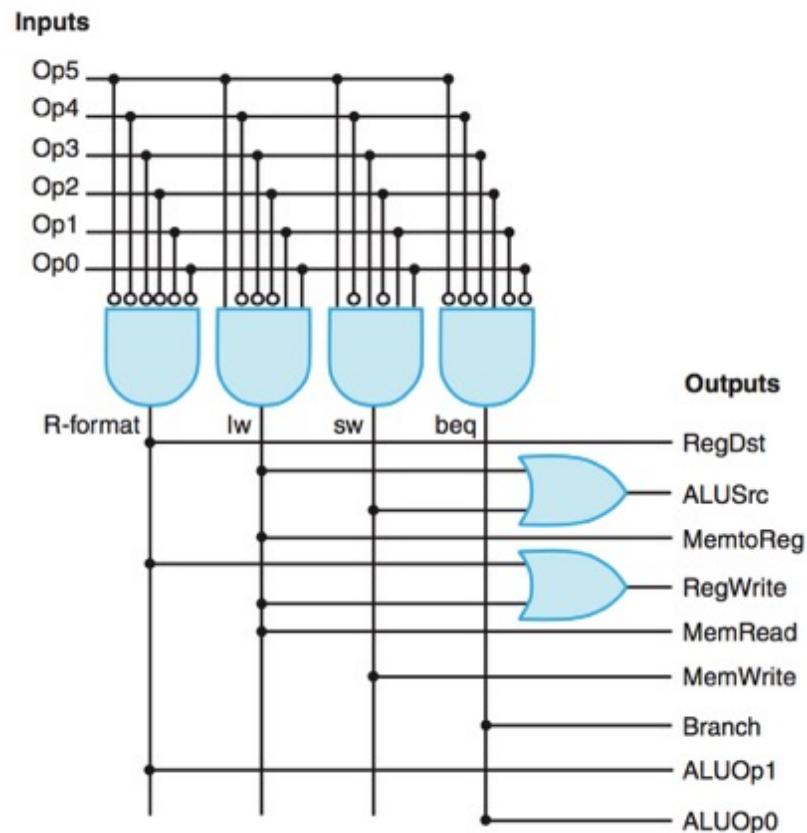
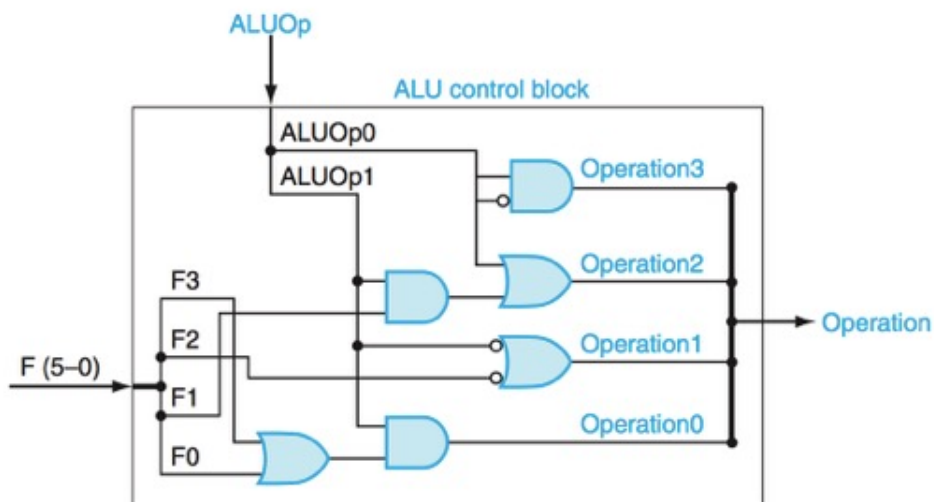


ICU: FSM

Hennessy & Patterson

Ch 10

Figure 10.2.3: The ALU control block generates the four ALU control bits, based on the function code and ALUOp bits (COD Figure D.2.3).



ICU: Microprogram

COMP122

ICU

ROM based State Machine

Instruction Register

- ❖ Plus: **simply** organized
- ❖ Minus: too **slow**
- ❖ Replacement: **FSM**

Opcode

Clock

Microprogram
Sequencer
(small FSM)

Clock

uPC

Microprogram
(in ROM)

uInstruction register

Clock

uInstruction

100
+

Control bits

ICU: Microprogram

Hennessy & Patterson

Ch 10

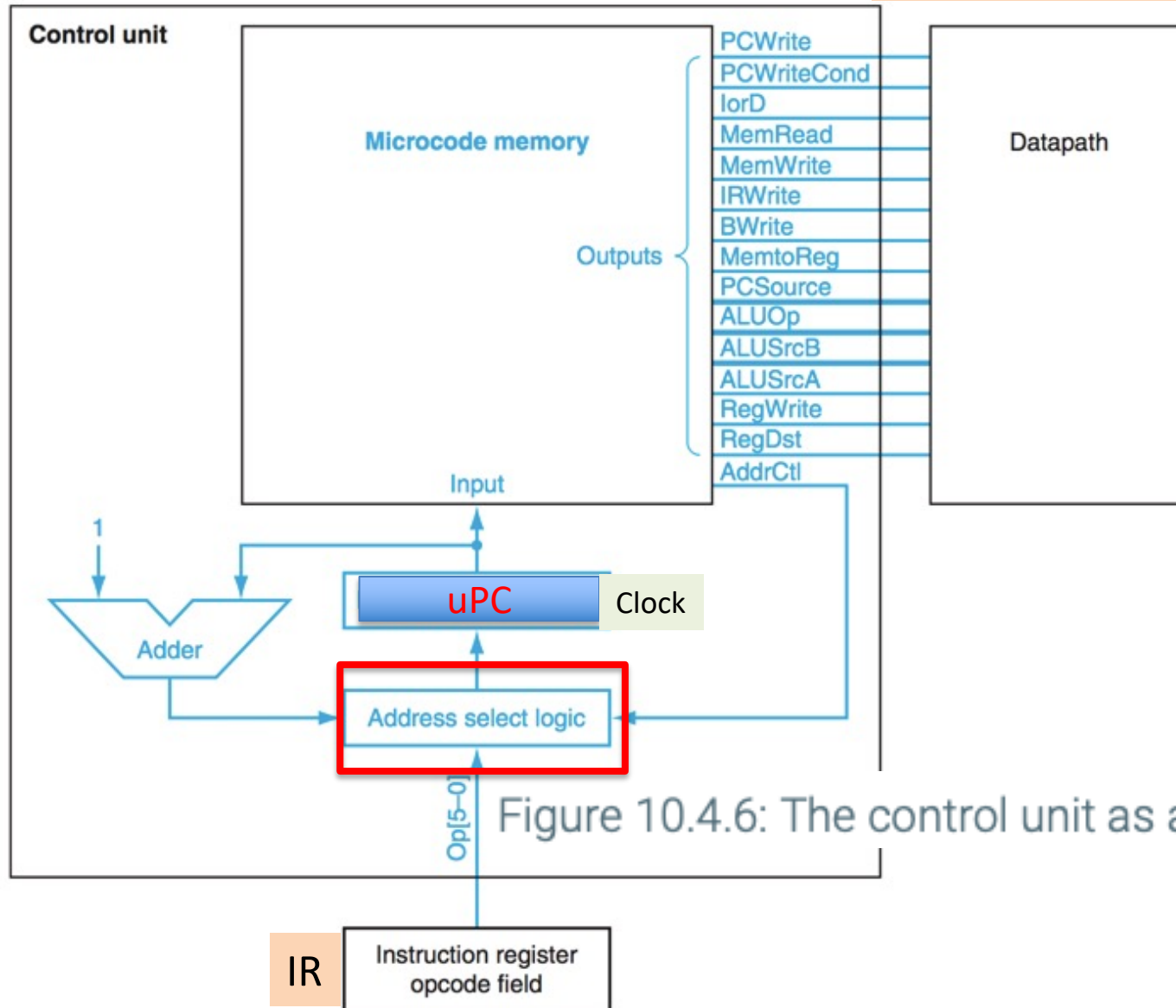
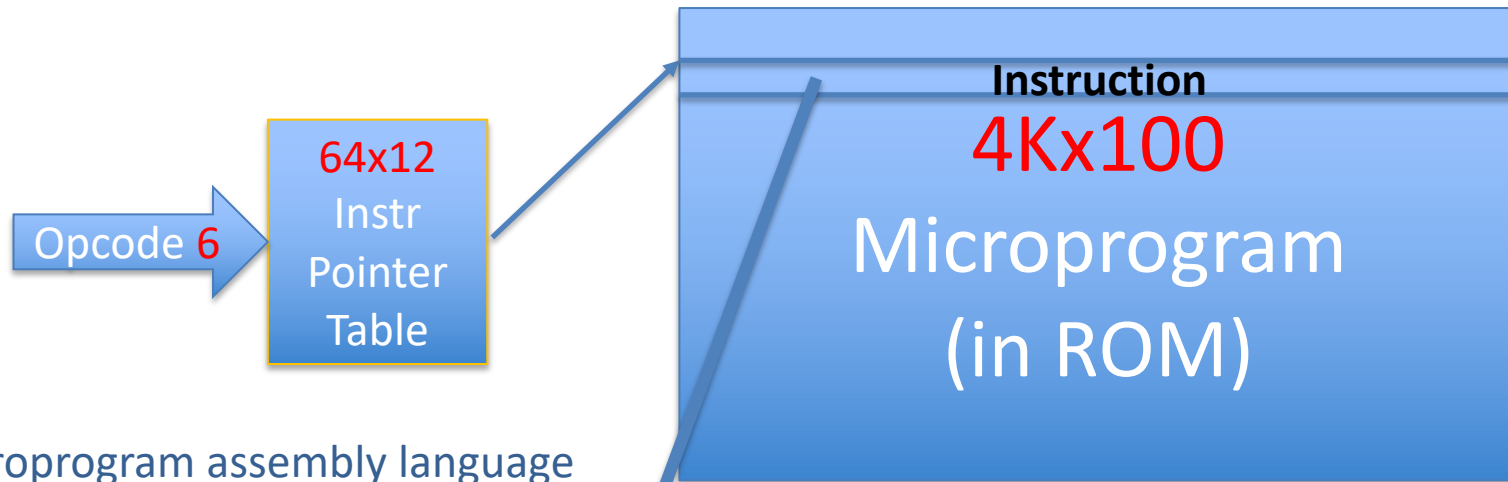


Figure 10.4.6: The control unit as a microcode

Microprogram

Address Select Logic as Table



Microprogram assembly language

ALU_OP=ADD, AMUX=GRA, BMUX=Shift → 100 Control bits

ALU OP	A MUX	B MUX	B SHIFT	GR dec A	GR dec B	GR dec D	GR mux
ADD	GR A	Shift	5	\$8	\$6	\$3	ALU
BR	PC	Offset	0	0	0	0	0
SUBI	GR A	Imm	0	\$4	0	\$5	ALU



Microprogramming

What is a hybrid microprogrammed control unit in computer architecture?



Jeff Drobman · just now

Lecturer at California State University, Northridge (2016–present)

in microprogramming, a “microinstruction” is a linear set of all the control bits sourced by the Instruction Control Unit (ICU) each clock cycle. it is typically 100–200 bits. in a pure hardware ICU, a complex logic state machine provides the microinstructions. in a pure “microprogrammed” ICU, each microinstruction is read out of a dedicated ROM memory addressed by a micro program counter (MPC). each machine instruction in the ISA has a corresponding microprogram subroutine to define it.

all early microprocessors were CISC and used microprogrammed ICU’s. once it was realized that reading from a ROM each cycle was a big bottleneck that slowed down the microprocessor, the new RISC architecture eliminated it and replaced it with a hardware ICU.

now microprogrammed ICU’s are back in a limited way in x86 micro-architecture in what is called by Intel “micro ops”. this is considered a hybrid ICU.

Computer Architecture

CPU

➤ See separate slide set

Organization:

Bit-slice

Bit-slice

Am2900



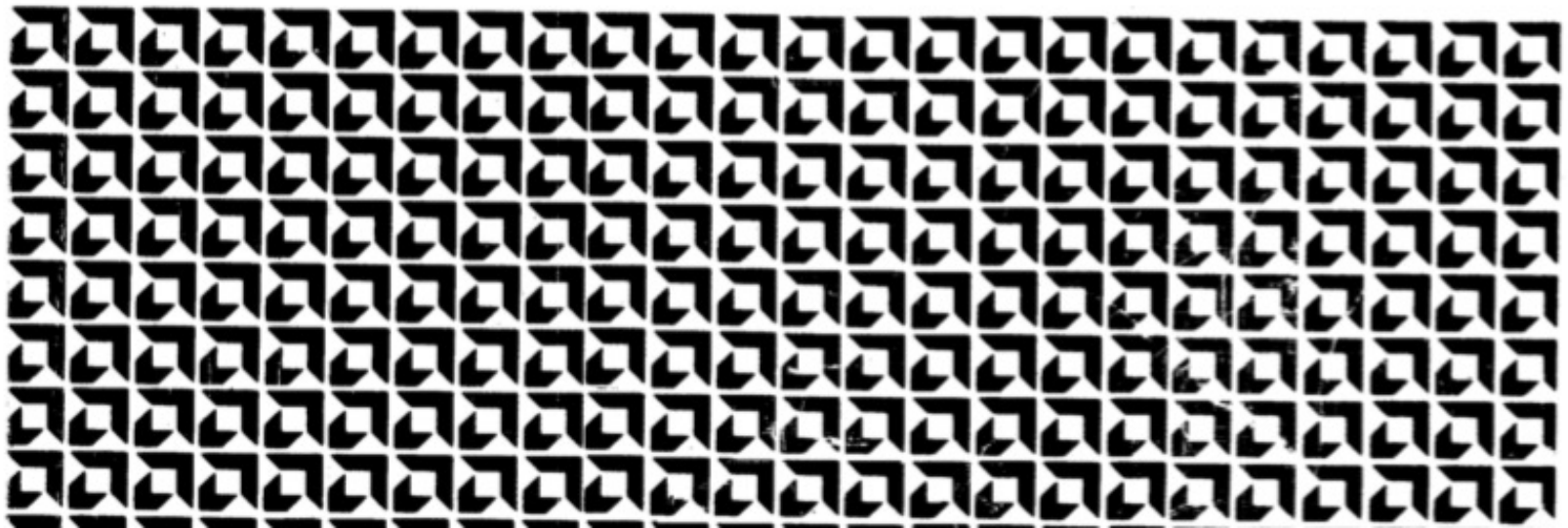
Am2900 Data Book

Am2900



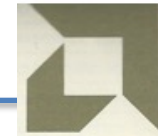
Advanced
Micro
Devices

The Am2900
Family
Data Book



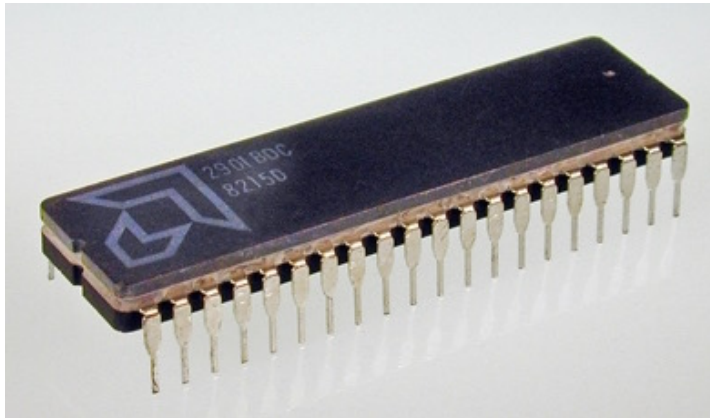
Am2900 Family

Bit-slice 1975-85



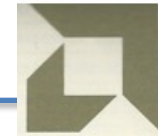
AMD 2901 bit-slice processor family includes 2901 and 2903 4-bit microprocessors slices, 2909 and 2911 microprogram sequencers, 2910 microprogram controller and other support chips. The 2901 processor consists of 16 4-bit registers, 4-bit ALU and associated decoding/multiplexing circuits. The ALU accepts 9-bit microinstructions that specify source operands, ALU function and the destination register. The 2901 ALU can perform 8 different functions (they are encoded into 3 bits within the microinstruction): addition, subtraction and logic operations. Multiple 2901 bit-slice processors could be combined together to build microprocessors with any data width (in 4 bits increments).

Enhanced version of 2901, AMD 2903 has 9 new special ALU functions used for implementation of multiplication, division and normalization operations. The number of arithmetic and logic ALU functions in 2903 was increased to 15.



AMD Am2903: 4-bit-slice ALU

Am2900 Family



Bit-slice 1975-85

Members of the Am2900 family [\[edit\]](#)

The Am2900 Family Data Book lists:^[22]

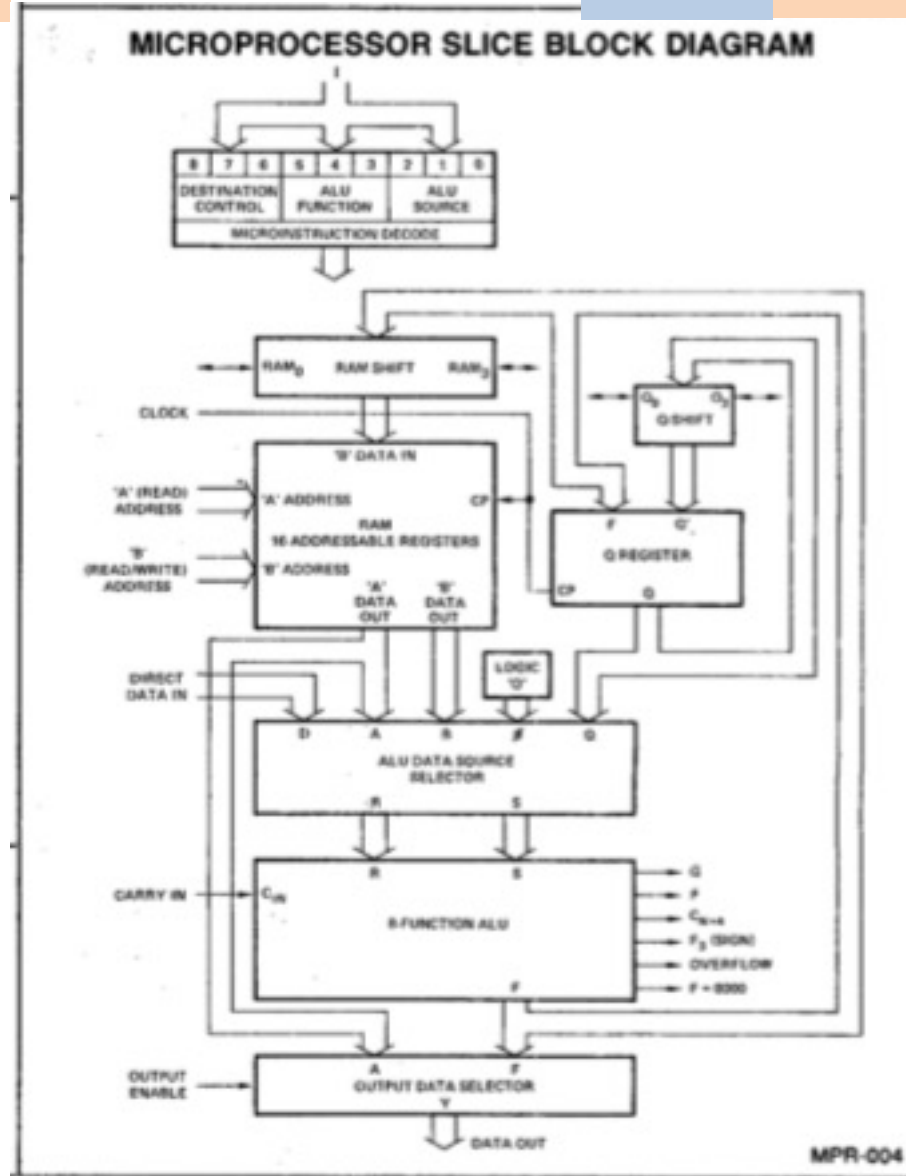
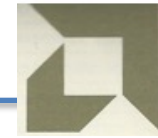
- Am2901 – 4-bit bit-slice [ALU](#) (1975)
- Am2902 – [Look-Ahead Carry](#) Generator
- Am2903 – 4-bit-slice ALU, with [hardware multiply](#)
- Am2904 – Status and Shift Control Unit
- Am2905 – Bus Transceiver
- Am2906 – Bus Transceiver with [Parity](#)
- Am2907 – Bus Transceiver with [Parity](#)
- Am2908 – Bus Transceiver with [Parity](#)
- Am2909 – 4-bit-slice address sequencer
- Am2910 – 12-bit address sequencer
- Am2911 – 4-bit-slice address sequencer
- Am2912 – Bus Transceiver
- Am2913 – Priority [Interrupt](#) Expander
- Am2914 – Priority [Interrupt](#) Controller
- Am2915 – Quad 3-State Bus Transceiver
- Am2916 – Quad 3-State Bus Transceiver
- Am2917 – Quad 3-State Bus Transceiver
- Am2918 – [Instruction Register](#), Quad D Register
- Am2919 – [Instruction Register](#), Quad Register
- Am2920 – Octal [D-Type Flip-Flop](#)
- Am2921 – 1-to-8 [Decoder](#)
- Am2922 – 8-Input [Multiplexer](#) (MUX)
- Am2923 – 8-Input [MUX](#)
- Am2924 – 3-Line to 8-Line [Decoder](#)
- Am2925 – [System Clock](#) Generator and Driver
- Am2926 – [Schottky](#) 3-State Quad Bus Driver
- Am2927/Am2928 – Quad 3-State Bus Transceiver
- Am2929 – Schottky 3-State Quad Bus Driver
- Am2930 – Main Memory Program Control
- Am2932 – Main Memory Program Control
- Am2940 – [Direct Memory Addressing \(DMA\)](#) Generator
- Am2942 – Programmable Timer/[Counter/DMA](#) Generator
- Am2946/Am2947 – Octal 3-State Bidirectional Bus Transceiver
- Am2948/Am2949 – Octal 3-State Bidirectional Bus Transceiver
- Am2950/Am2951 – 8-bit Bidirectional I/O Ports
- Am2954/Am2955 – Octal Registers
- Am2956/Am2957 – Octal Latches
- Am2958/Am2959 – Octal [Buffers](#)/Line Drivers/Line Receivers
- Am2960 – Cascadable 16-bit Error Detection and Correction Unit
- Am2961/Am2962 – 4-bit Error Correction Multiple Bus Buffers
- Am2964 – Dynamic Memory Controller
- Am2965/Am2966 – Octal Dynamic Memory Driver

2901 Block Diagram

Am2900

Bit-slice

1975-85



2901 8 ALU Ops

Am2900

Mnemonic	MICRO CODE				ALU SOURCE OPERANDS	
	I ₂	I ₁	I ₀	Octal Code	R	S
AQ	L	L	L	0	A	Q
AB	L	L	H	1	A	B
ZQ	L	H	L	2	O	Q
ZB	L	H	H	3	O	B
ZA	H	L	L	4	O	A
DA	H	L	H	5	D	A
DQ	H	H	L	6	D	Q
DZ	H	H	H	7	D	O

Figure 2. ALU Source Operand Control.

Mnemonic	MICRO CODE				ALU Function	SYMBOL
	I ₅	I ₄	I ₃	Octal Code		
ADD	L	L	L	0	R Plus S	$R + S$
SUBR	L	L	H	1	S Minus R	$S - R$
SUBS	L	H	L	2	R Minus S	$R - S$
OR	L	H	H	3	R OR S	$R \vee S$
AND	H	L	L	4	R AND S	$R \wedge S$
NOTRS	H	L	H	5	\bar{R} AND S	$\bar{R} \wedge S$
EXOR	H	H	L	6	R EX OR S	$R \veebar S$
EXNOR	H	H	H	7	R EX NOR S	$\overline{R \veebar S}$

Figure 3. ALU Function Control.

Mnemonic	MICRO CODE				RAM FUNCTION		Q-REG. FUNCTION		Y OUTPUT	RAM SHIFTER		Q SHIFTER	
	I ₈	I ₇	I ₆	Octal Code	Shift	Load	Shift	Load		RAM ₀	RAM ₃	Q ₀	Q ₃
QREG	L	L	L	0	X	NONE	NONE	F → Q	F	X	X	X	X
NOP	L	L	H	1	X	NONE	X	NONE	F	X	X	X	X
RAMA	L	H	L	2	NONE	F → B	X	NONE	A	X	X	X	X
RAMF	L	H	H	3	NONE	F → B	X	NONE	F	X	X	X	X
RAMQD	H	L	L	4	DOWN	F/2 → B	DOWN	Q/2 → Q	F	F ₀	IN ₃	Q ₀	IN ₃
RAMD	H	L	H	5	DOWN	F/2 → B	X	NONE	F	F ₀	IN ₃	Q ₀	X
RAMQU	H	H	L	6	UP	2F → B	UP	2Q → Q	F	IN ₀	F ₃	IN ₀	Q ₃
RAMU	H	H	H	7	UP	2F → B	X	NONE	F	IN ₀	F ₃	X	Q ₃

✗ DON'T USE

□ DISABLE FEN

△ ENABLE FEN

X = Don't care. Electrically, the shift pin is a TTL input internally connected to a three-state output which is in the high-impedance state

B = Register Addressed by B inputs.

UP is toward MSB, DOWN is toward LSB.

Figure 4. ALU Destination Control.

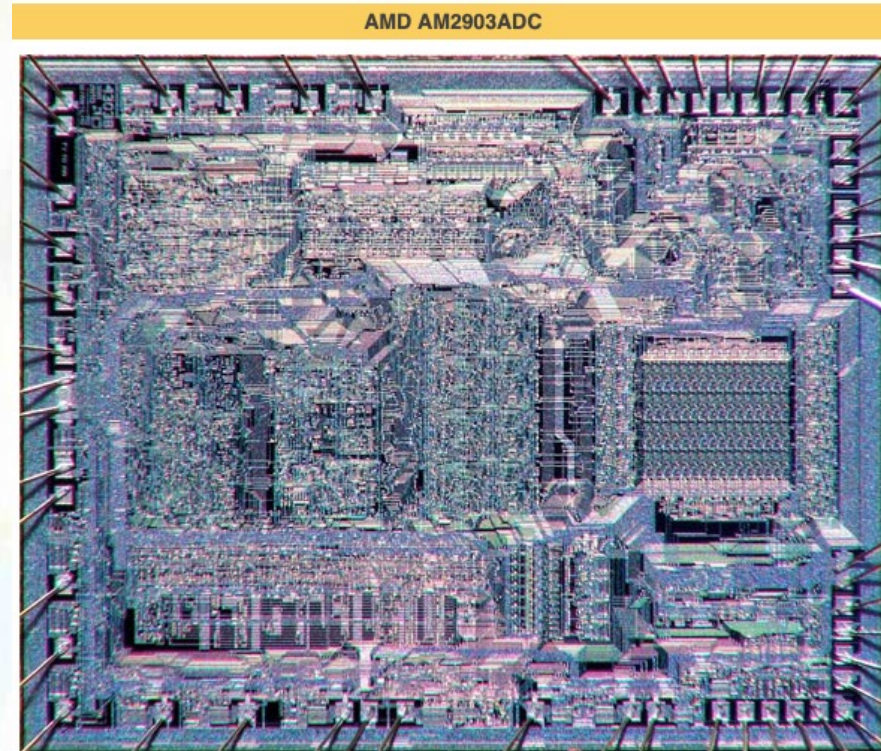
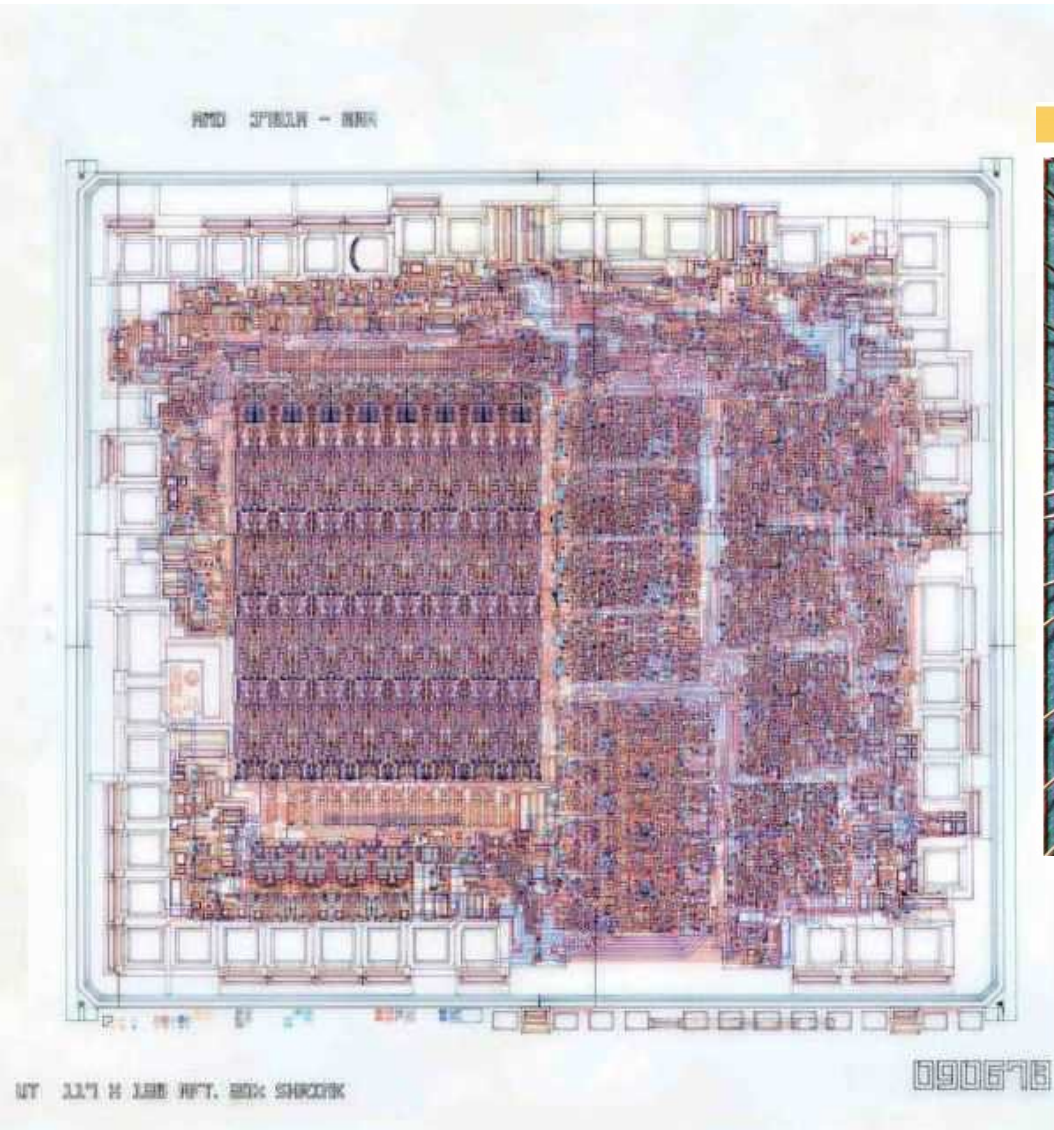
Am2900



System Block Diagram

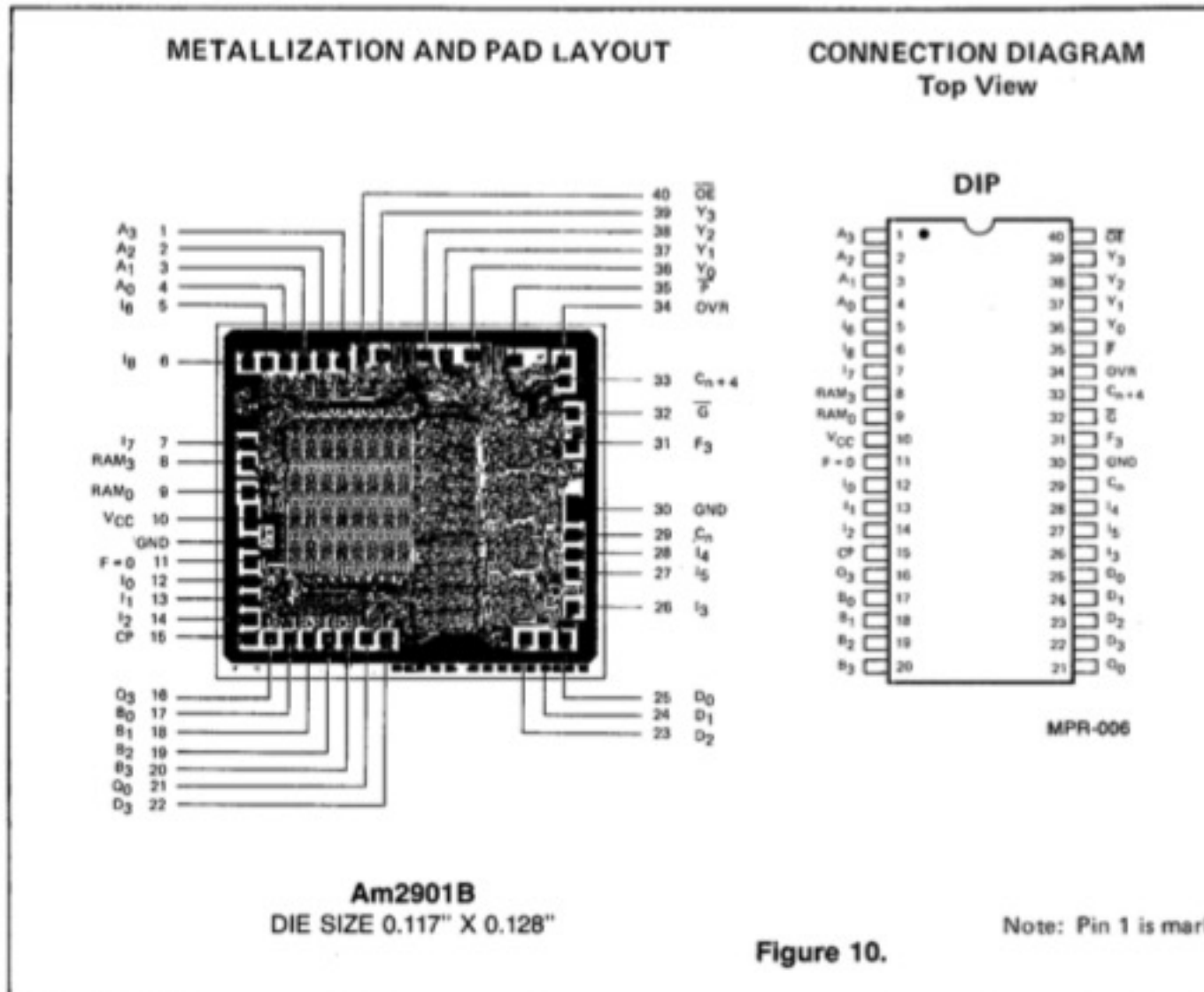
Am2901/3 4-bit MPU

Bit-slice 1975-85



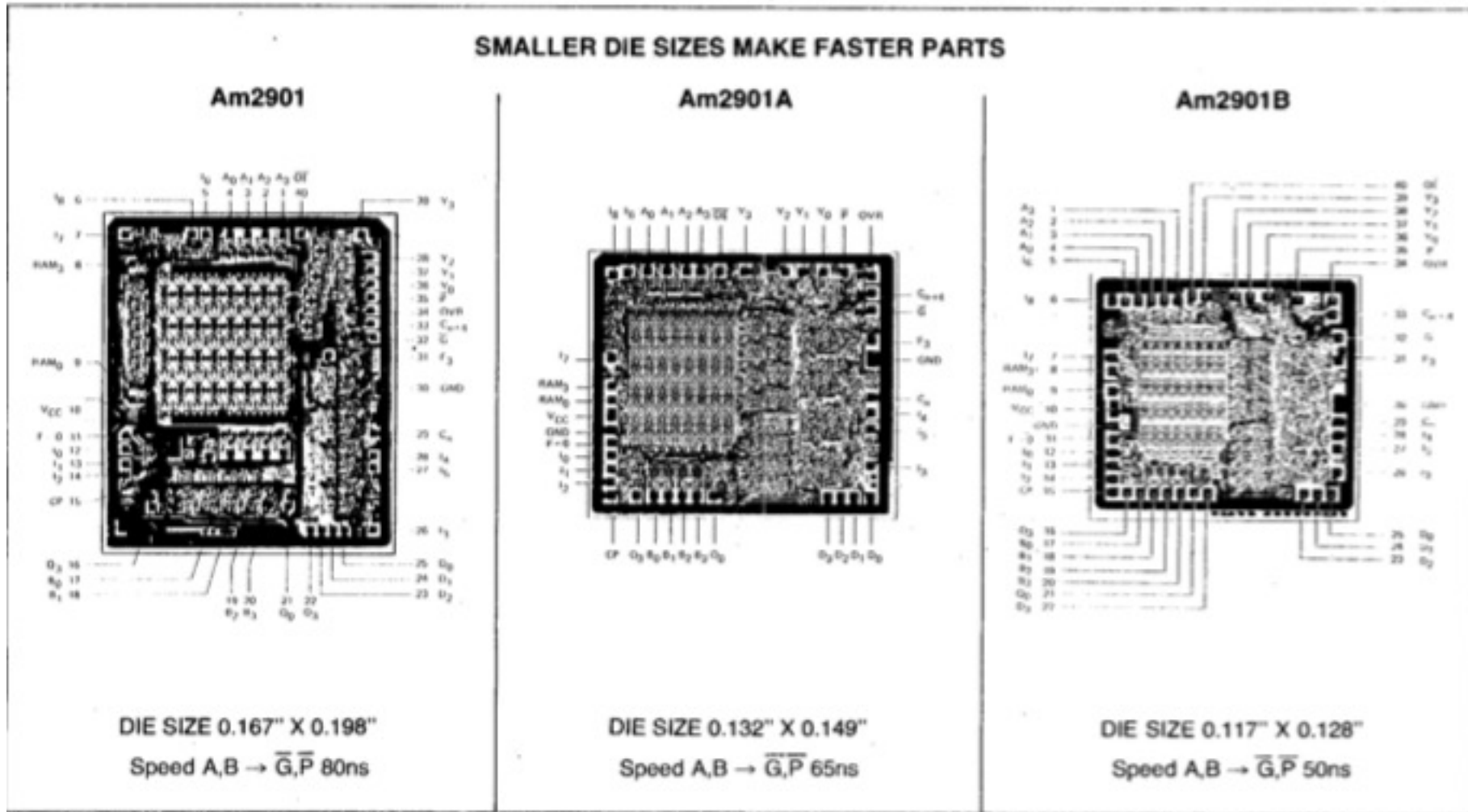
2901 Chip

Am2900



2901-A-B Die

Am2900



Computer Org: ICU

RISC Pipelines

CISC vs RISC:

Complex/Reduced Instruction Set Architecture

❖ Microprocessor History

- 1971-85: **CISC** (8/16-bit)
 - ✧ Intel i4004 (4-bit)
 - ✧ Intel i8008 (8-bit) → i8080 → i8085, Z80 → i8086 (16-bit) → “x86”
 - ✧ Motorola 6800 (8-bit) → 6502 → 68000 (16-bit)
 - ✧ IBM PC used i8088 (8/16-bit) in 1981 → i80n86 (“x86”) → *Pentiums*
- 1985-2000: **RISC** – (32/64-bit)
 - ✧ **SPARC*** (UC Berkeley → Sun/Oracle)
 - ✧ MIPS* (Stanford)
 - ✧ PowerPC (Motorola/IBM)
 - ✧ AMD 29K
 - ✧ Intel i960
 - ✧ ARM*

*still exist

RISC:

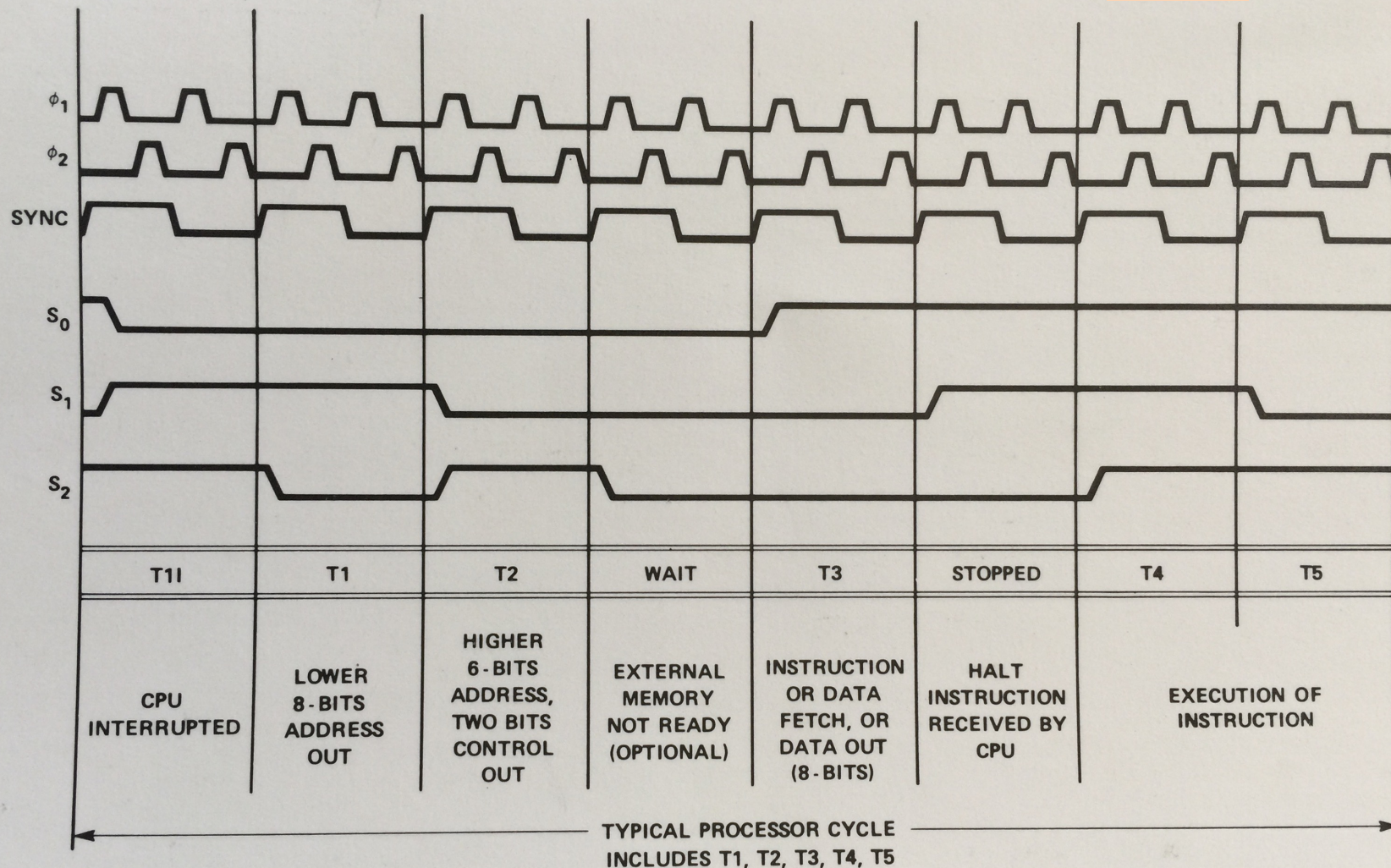
Reduced Instruction Set Architecture

❖ Key Architecture of RISC

- Reduced ISA: small set of instructions
- Fast execution: single cycle only
- Reduced impact of memory
 - ✧ No microprogram (key change)
 - Instructions scale to vertical microinstructions (single-cycle)
 - eliminates ~30% chip area
 - ✧ LOAD-STORE (only) memory references
 - ✧ Full general register sets
 - ✧ Cache memory
 - On-chip
 - Multi-level
 - Harvard architecture – separate I and D
- Pipelining
 - ✧ 4 or 5 stages
 - ✧ Interlocks
 - Hardware (SPARC, 29K)
 - Software (MIPS): compiler manages pipeline scheduling

CISC Instruction Cycle

MCS-8



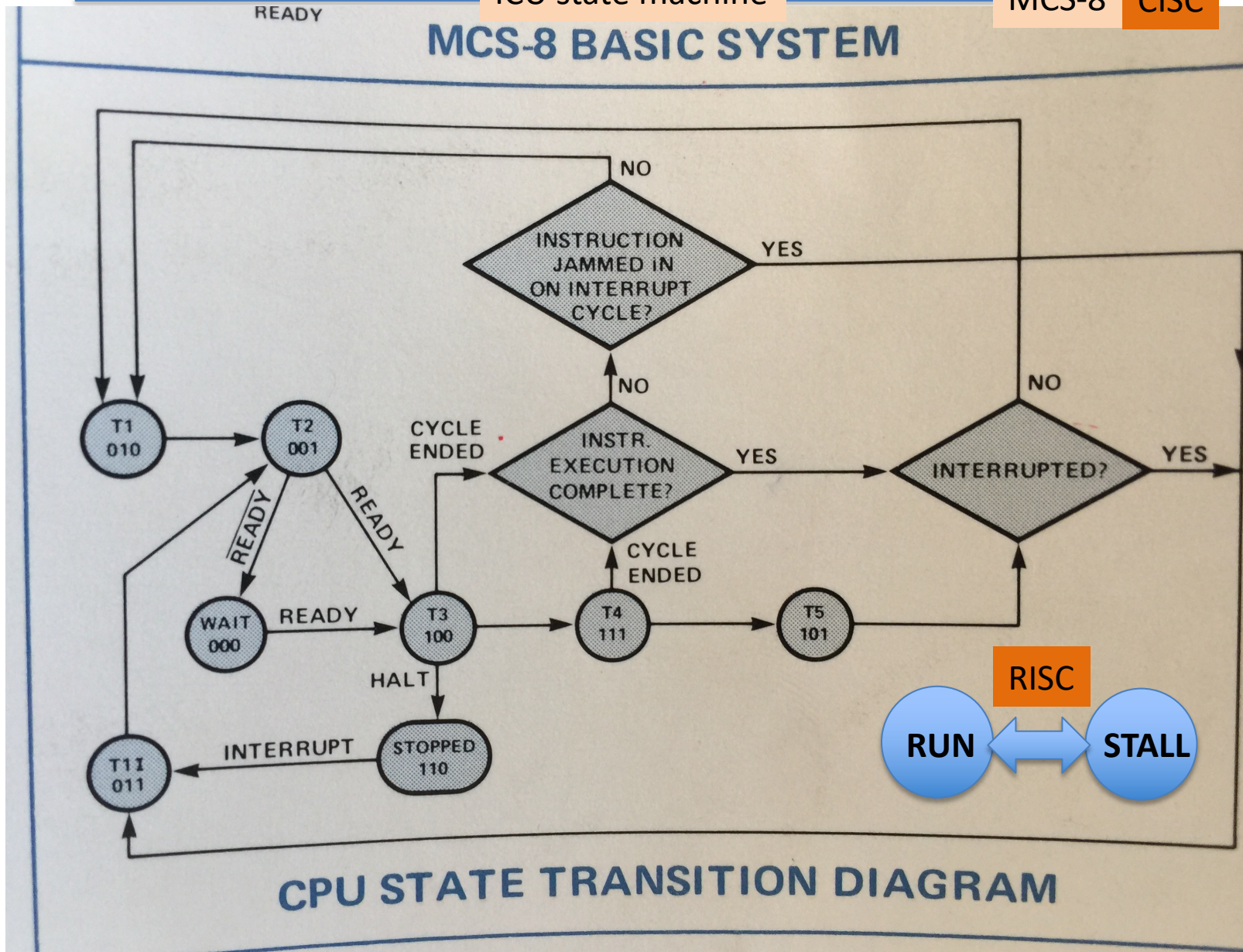
MCS-8 BASIC INSTRUCTION CYCLE

CISC State Diagram

ICU state machine

MCS-8

CISC



Instruction Cycles

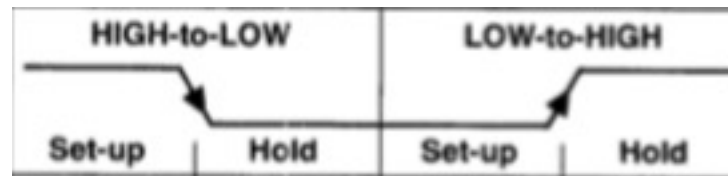
Clock Sync

Clocks ⇔ Cycles

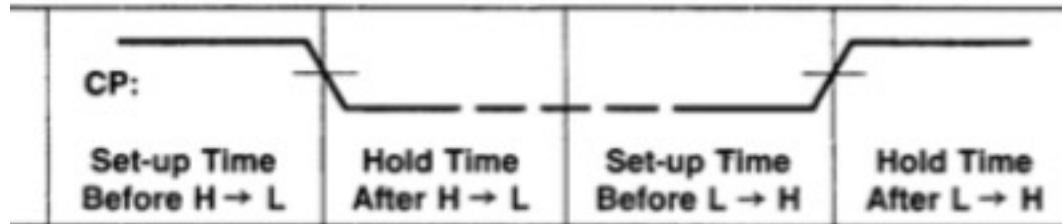
clock speed is normally the frequency that a CPU operates at, and inversely, defines the *clock period* or *cycle time*.



CPI (*clocks per instruction*) is the average number of *cycles* it takes to execute an instruction – per a given *instruction stream*.



Set-up and Hold Times Relative to Clock (CP) Input.

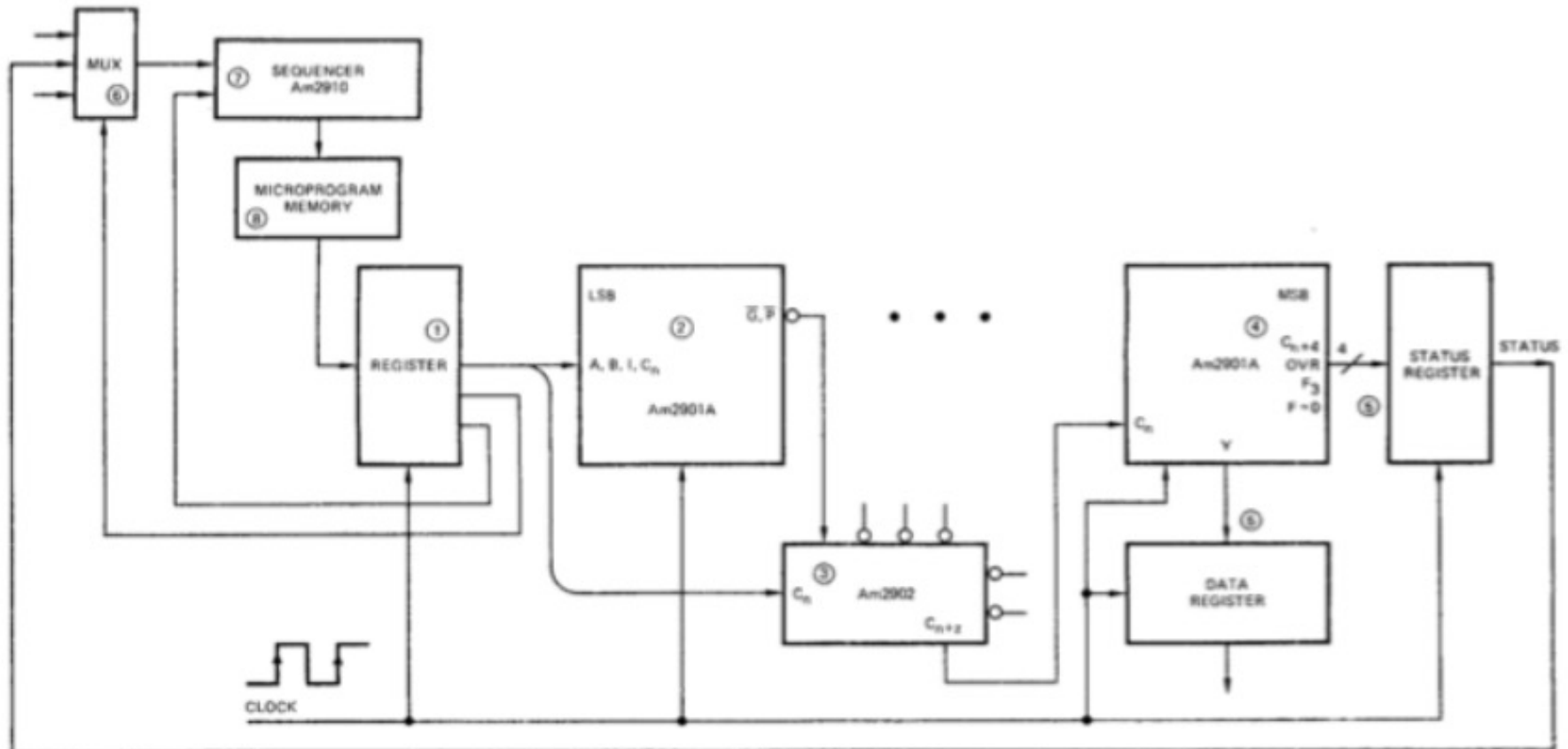


Cycle Times

Am2900

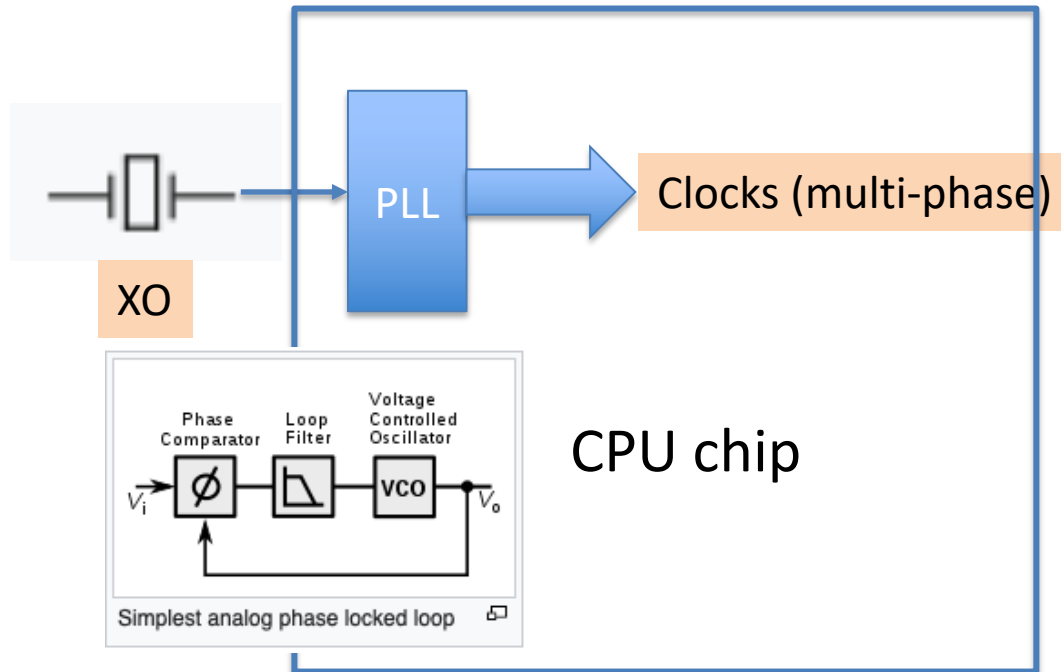
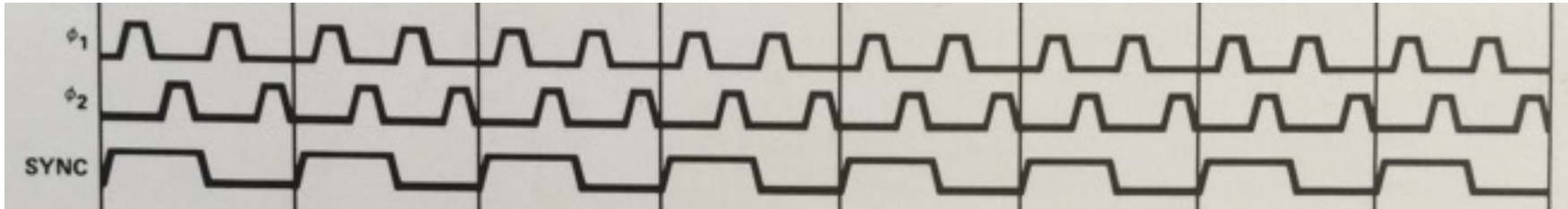
MINIMUM CYCLE TIME CALCULATIONS FOR 16-BIT SYSTEMS

Speeds used in calculations for parts other than Am2901B are representative for available MSI parts.

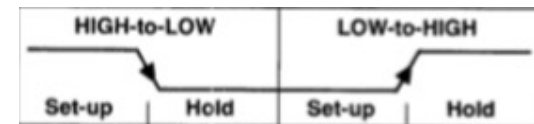


CPU Chip Clock Source

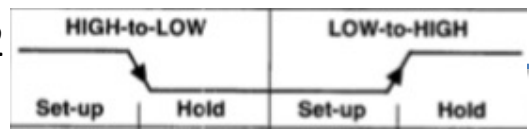
2-phase clock



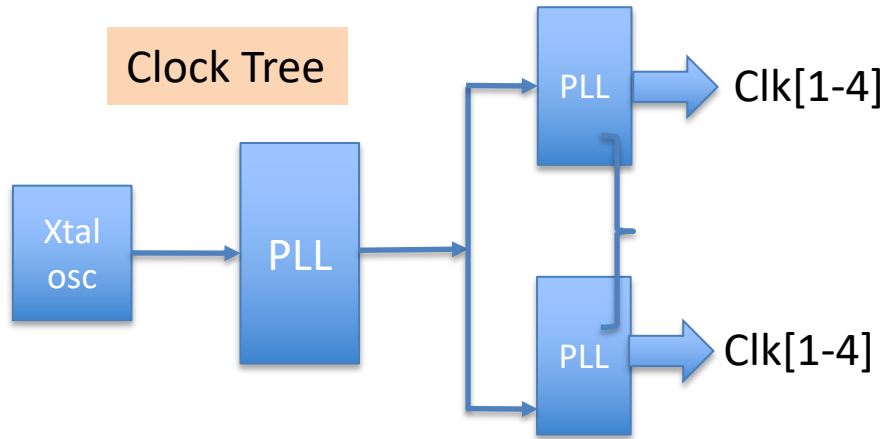
PLL will *multiply*
300 MHz xtal freq
Up to 4 GHz



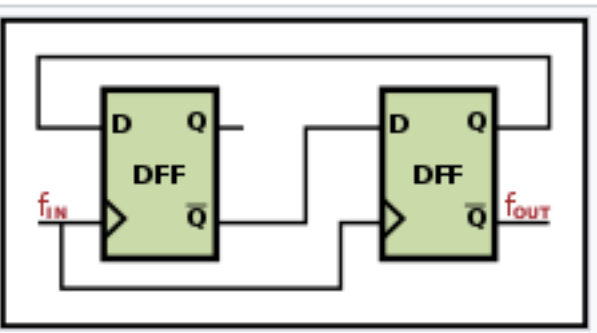
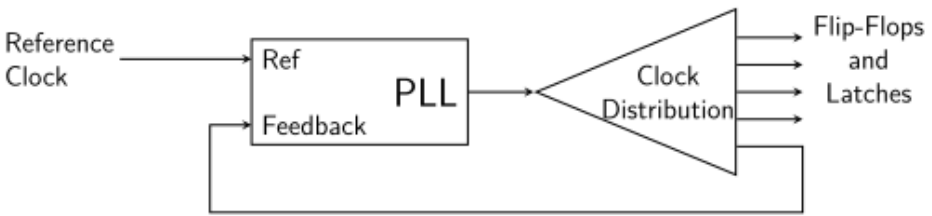
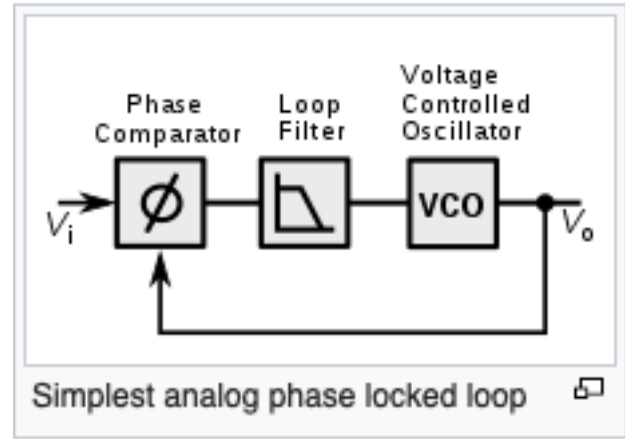
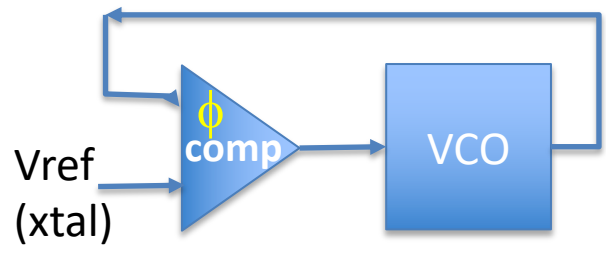
Clock Gen



Clock Tree



PLL= *Phase* Locked Loop



An example digital divider (by 4) for use in the feedback path of a multiplying PLL

Freq divider (by 4)

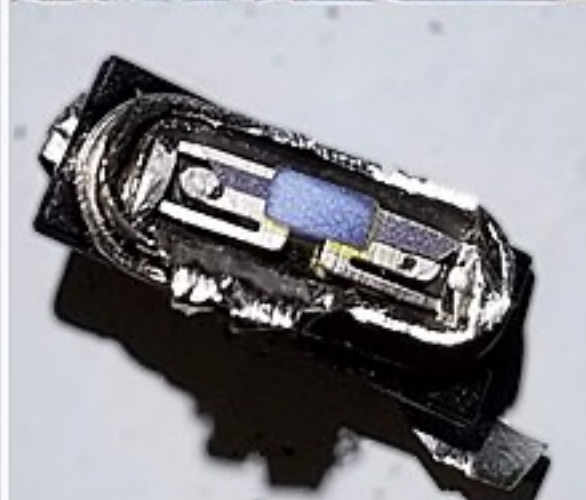


Clock Source: Xtal Osc

Crystal oscillator



Inside a modern DIP package quartz crystal oscillator module. It includes a ceramic PCB base, oscillator, divider chip (/8), bypass capacitor, and an AT cut crystal.



Internals of a quartz crystal.



Cluster of natural quartz crystals



A synthetic quartz crystal grown by the [hydrothermal synthesis](#), about 19 cm long and weighing about 127 g

Clock Source: Xtal Osc

Crystal oscillator types and their abbreviations:

- **ATCXO** — [Analog temperature controlled crystal oscillator](#)
- **CDXO** — Calibrated dual crystal oscillator
- **DTCXO** — Digital temperature compensated crystal oscillator
- **EMXO** — Evacuated miniature crystal oscillator
- **GPSDO** — [Global positioning system disciplined oscillator](#)
- **MCXO** — [Microcomputer-compensated crystal oscillator](#)
- **OCVCXO** — [oven-controlled voltage-controlled crystal oscillator](#)
- **OCXO** — [Oven-controlled crystal oscillator](#)
- **RbXO** — [Rubidium](#) crystal oscillators (RbXO), a crystal oscillator (can be an save power
- **TCVCXO** — Temperature-compensated [voltage-controlled crystal oscillator](#)
- **TCXO** — Temperature-compensated crystal oscillator
- **TMXO** — Tactical miniature crystal oscillator^[67]
- **TSXO** — Temperature-sensing crystal oscillator, an adaptation of the TCXO
- **VCTCXO** — Voltage-controlled temperature-compensated crystal oscillator
- **VCXO** — Voltage-controlled crystal oscillator

CISC/RISC Pipelines

Non pipelined

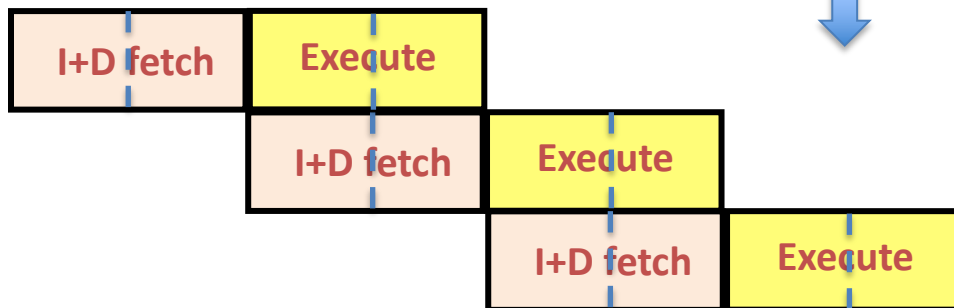
4-8 cycles per I
i8008/M6800



One cycle

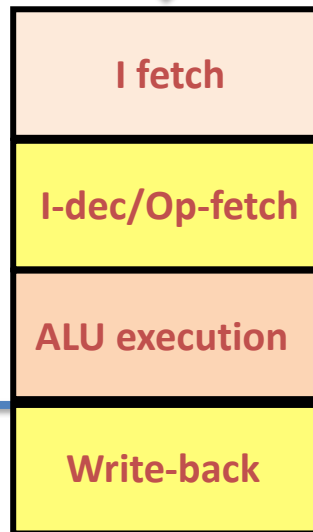
2-4 cycles per I i8088/M68000

CISC Pipeline 2-stage



One cycle

Instructions



Data



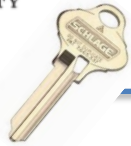
RISC Pipeline 4/5-stage

R3000/SPARC/i960/29K/PPC

One cycle

1 cycle per I

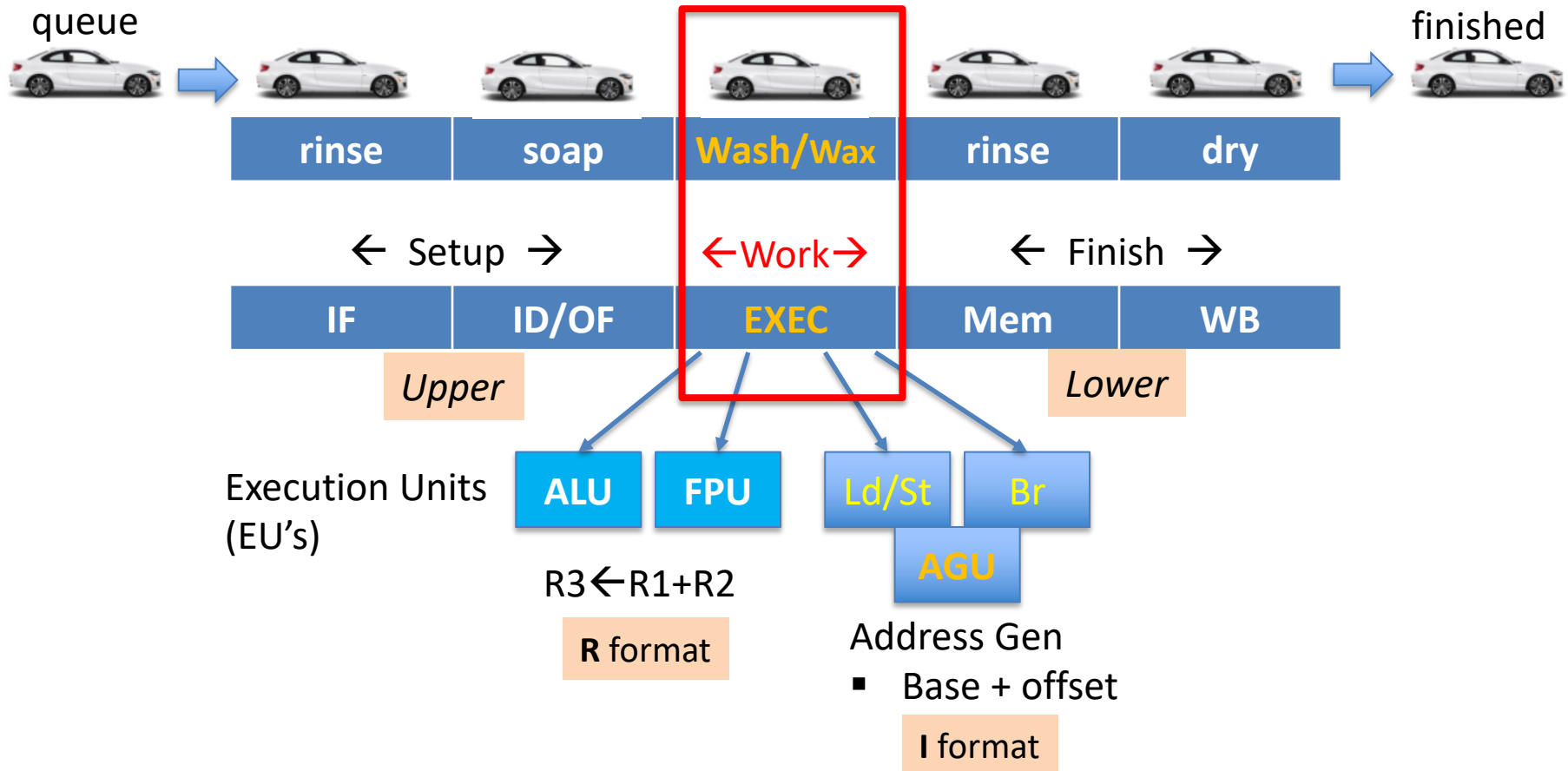
Hardware Interlock
or
Delay Slot (NOP)
(for LOAD, BR)



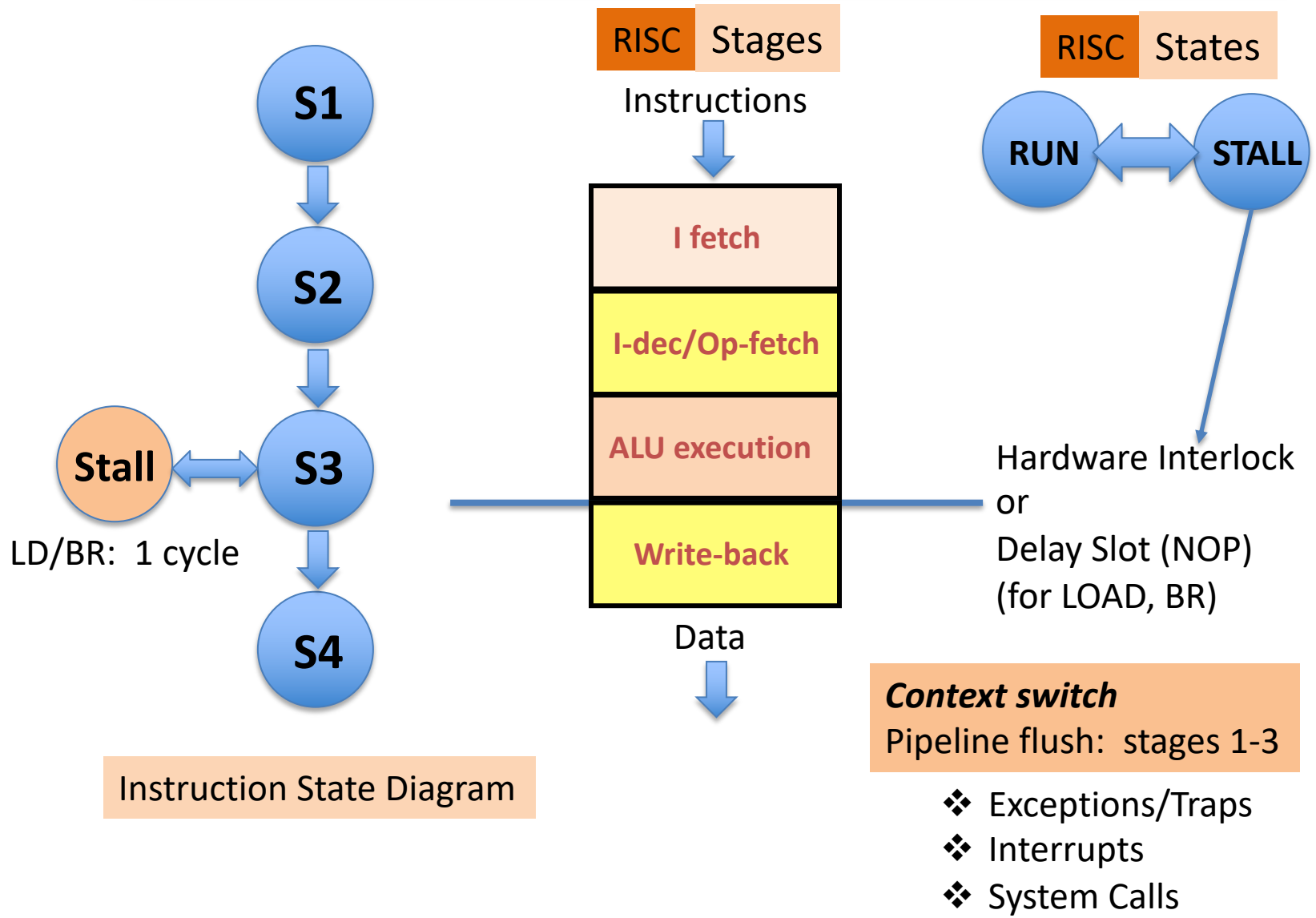
MIPS RISC Pipeline

5 Stages

Each stage takes only 1/5 of instruction cycle: **clock F \Rightarrow 5x**



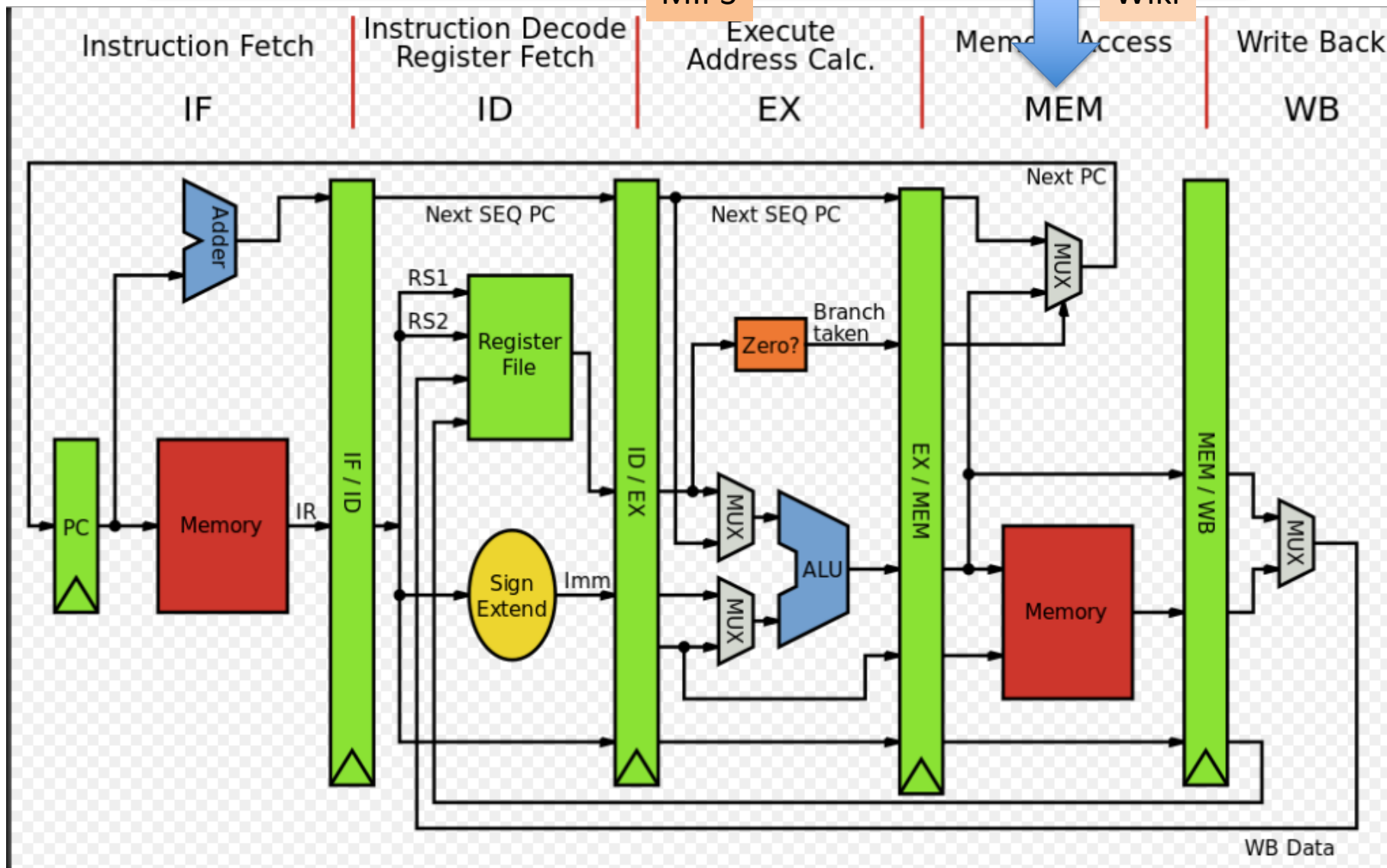
RISC Pipelines: Stages/States



MIPS Pipelined Org

MIPS

Wiki



MIPS, showing the five stages (instruction fetch, instruction decode, execute, memory access and write back).

Additional Material

JTAG

IEEE Joint Test Action Group

Boundary Scan

Boundary scan

From Wikipedia, the free encyclopedia
(Redirected from [JTAG boundary scan](#))

Boundary scan is a method for testing interconnects (wire lines) on [printed circuit boards](#) or sub-blocks inside an [integrated circuit](#). Boundary scan is also widely used as a debugging method to watch integrated circuit pin states, measure voltage, or analyze sub-blocks inside an integrated circuit.

The [Joint Test Action Group](#) (JTAG) developed a specification for boundary scan testing that was standardized in 1990 as the [IEEE Std. 1149.1-1990](#). In 1994, a supplement that contains a description of the [Boundary Scan Description Language](#) (BSDL) was added which describes the boundary-scan logic content of IEEE Std 1149.1 compliant devices. Since then, this standard has been adopted by electronic device companies all over the world. Boundary scan is now mostly synonymous with JTAG.^{[1][2]}

Debugging [[edit](#)]

The boundary scan architecture also provides functionality which helps [developers](#) and [engineers](#) during development stages of an embedded system. A JTAG Test Access Port (TAP) can be turned into a low-speed [logic analyzer](#).

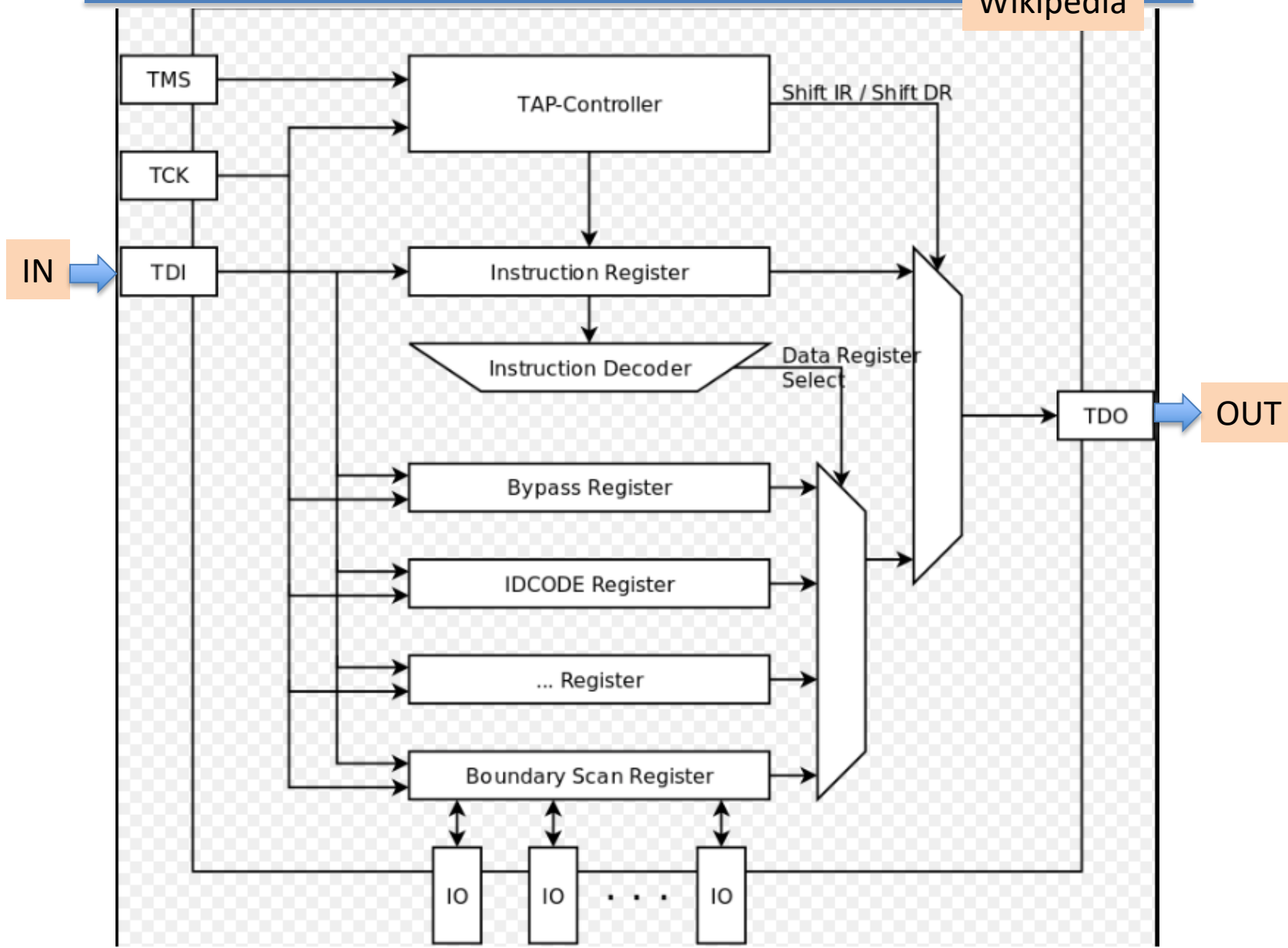
History [[edit](#)]

James B. Angell at Stanford University proposed serial testing.^[4]

IBM developed [level-sensitive scan design](#) (LSSD).^{[5][6]}

JTAG

Wikipedia



On-chip infrastructure [\[edit \]](#)

To provide the boundary scan capability, IC vendors add additional logic to each of their devices, including *scan cells* for each of the external traces. These cells are then connected together to form the external boundary scan shift register (BSR), and combined with [JTAG](#) Test Access Port (TAP) controller support comprising four (or sometimes more) additional pins plus control circuitry.

Some TAP controllers support [scan chains](#) between on-chip logical design blocks, with JTAG instructions which operate on those internal scan chains instead of the BSR. This can allow those integrated components to be tested as if they were separate chips on a board. On-chip debugging solutions are heavy users of such internal scan chains.

These designs are part of most [Verilog](#) or [VHDL](#) libraries. Overhead for this additional logic is minimal, and generally is well worth the price to enable efficient testing at the board level.

For normal operation, the added boundary scan latch cells are set so that they have no effect on the circuit, and are therefore effectively invisible. However, when the circuit is set into a test mode, the latches enable a data stream to be shifted from one latch into the next. Once a complete data word has been shifted into the circuit under test, it can be latched into place so it drives external signals. Shifting the word also generally returns the input values from the signals configured as inputs.

Test mechanism [\[edit \]](#)

As the cells can be used to force data into the board, they can set up test conditions. The relevant states can then be fed back into the test system by clocking the data word back so that it can be analyzed.

By adopting this technique, it is possible for a test system to gain test access to a board. As most of today's boards are very densely populated with components and tracks, it is very difficult for test systems to physically access the relevant areas of the board to enable them to test the board. Boundary scan makes access possible without always needing physical probes.

In modern chip and board design, [Design For Test](#) is a significant issue, and one common design artifact is a set of boundary scan test vectors, possibly delivered in [Serial Vector Format](#) (SVF) or a similar interchange format.

JTAG test operations [\[edit\]](#)

Devices communicate to the world via a set of input and output pins. By themselves, these pins provide limited visibility into the workings of the device. However, devices that support boundary scan contain a shift-register cell for each signal pin of the device. These registers are connected in a dedicated path around the device's boundary (hence the name). The path creates a virtual access capability that circumvents the normal inputs and provides direct control of the device and detailed visibility at its outputs.^[3] The contents of the boundary scan are usually described by the manufacturer using a part-specific [BSDL](#) file.

Among other things, a BSDL file will describe each digital signal exposed through pin or ball (depending on the chip packaging) exposed in the boundary scan, as part of its definition of the Boundary Scan Register (BSR). A description for two balls might look like this:

```
"541 (bc_1,          *, control, 1)," &
"542 (bc_1,    GPIO51_ATACS1, output3, X, 541, 1, Z)," &
"543 (bc_1,    GPIO51_ATACS1, input, X)," &
"544 (bc_1,          *, control, 1)," &
"545 (bc_1,    GPIO50_ATACS0, output3, X, 544, 1, Z)," &
"546 (bc_1,    GPIO50_ATACS0, input, X)," &
```

That shows two balls on a mid-size chip (the boundary scan includes about 620 such lines, in a 361-ball [BGA](#) package), each of which has three components in the BSR: a control configuring the ball (as input, output, what drive level, pullups, pulldowns, and so on); one type of output signal; and one type of input signal.

There are JTAG instructions to **SAMPLE** the data in that boundary scan register, or **PRELOAD** it with values.

During testing, I/O signals enter and leave the chip through the boundary-scan cells. Testing involves a number of test vectors, each of which drives some signals and then verifies that the responses are as expected. The boundary-scan cells can be configured to support external testing for interconnection between chips (**EXTEST** instruction) or internal testing for logic within the chip (**INTEST** instruction).



Drazen Zoric · [Follow](#)

Lives in Cork, Ireland · C

JTAG



Jeff Drobman · Just now

very good, but you did not mention that the purpose of JTAG is to provide a test port. select chip internal bits are in a test chain which are serially shifted in/out of the chip via the 4-pin JTAG port.

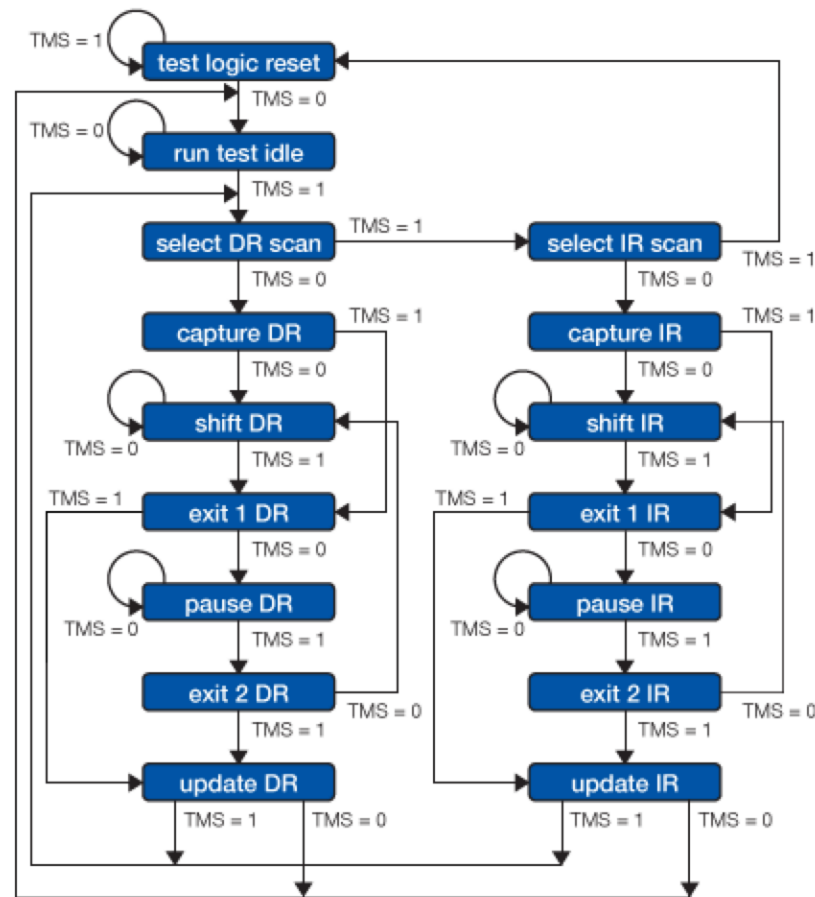


Drazen Zoric · Follow

Lives in Cork, Ireland · 6

JTAG

JTAG is based on state-machine implemented in hardware and debugger for any operation must walk among states. Here one example:



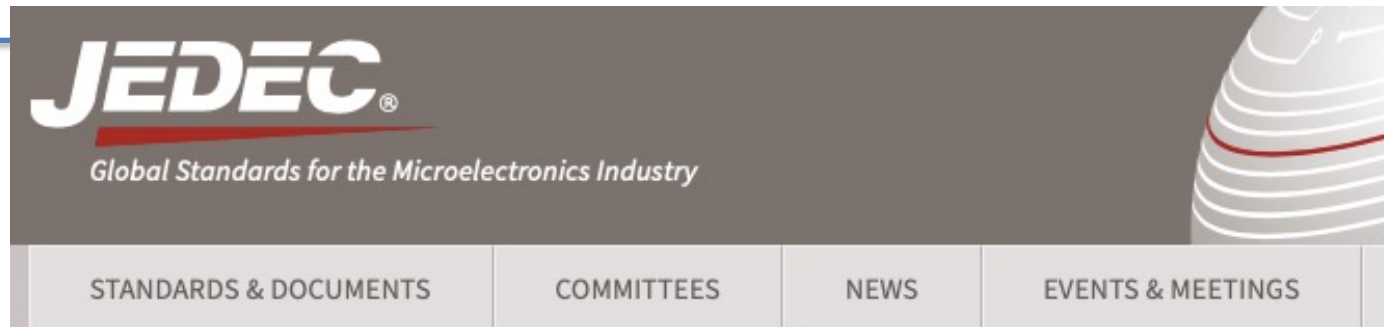
Jeff Drobman · Just now

very good, but you did not mention that the purpose of JTAG is to provide a test port. select chip internal bits are in a test chain which are serially shifted in/out of the chip via the 4-pin JTAG port.

Additional Material

JEDEC

JEDEC (EIA)



GRAPHICS DOUBLE DATA RATE 6 (GDDR6) SGRAM STANDARD

JESD250B

Published: Nov 2018

This document defines the Graphics Double Data Rate 6 (GDDR6) Synchronous Graphics Random Access Memory (SGRAM) specification, including features, functionality, package, and pin assignments. The purpose of this Specification is to define the minimum set of requirements for 8 Gb through 16 Gb x16 dual channel GDDR6 SGRAM devices. System designs based on the required aspects of this standard will be supported by all GDDR6 SGRAM vendors providing compatible devices. Some aspects of the GDDR6 standard such as AC timings and capacitance values were not standardized. Some features are optional and therefore may vary among vendors. In all cases, vendor data sheets should be consulted for specifics. This document was created based on some aspects of the GDDR5 Standard (JESD212). Item 1836.99D.

JEDEC (EIA)

JEDEC History

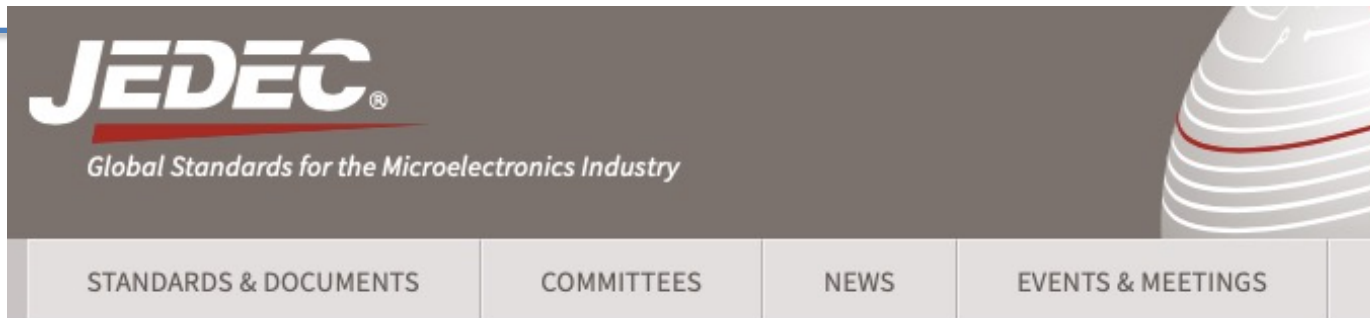
In 1924, the Radio Manufacturers Association (which later became the Electronic Industries Association) was established. In 1944, the Radio Manufacturers Association and the National Electronic Manufacturers Association established the Joint Electron Tube Engineering Council (JETEC), which was responsible for assigning and coordinating type numbers of electron tubes. As the radio industry expanded into the emerging field of electronics, various divisions of the EIA, including JETEC, began to function as semi-independent membership groups. The Council expanded its scope to include solid state devices, and by 1958 the organization was renamed the Joint Electron Device Engineering Council (JEDEC) – one council for tubes and one for semiconductors.

Timeline

- [Pre-1960s](#)
- [1960s](#)
- [1970s](#)
- [1980s](#)
- [1990s](#)
- [2000s](#)
- [2010s](#)

JEDEC initially functioned within the engineering department of EIA where its primary activity was to develop and assign part numbers to devices. Over the next 50 years, JEDEC's work expanded into developing test methods and product standards that proved vital to the development of the semiconductor industry. Among the landmark standards that have come from JEDEC committees are:

JEDEC (EIA)



Why JEDEC Standards Matter

JEDEC committees develop open standards, which are the basic building blocks of the digital economy and form the bedrock on which healthy, high-volume markets are built. For example, JEDEC semiconductor memory standards - from dynamic RAM chips and memory modules to DDR synchronous DRAM and flash components - have enabled huge markets in PCs, servers, digital cameras, MP3 players, smart phones, automotive and HDTV, to name just a few.

Standards enable innovation, serving to commoditize components by lowering their prices while maintaining quality and reliability. This leads suppliers to compete more vigorously on innovative features and gives buyers more variety and a broader selection. The end result is a much larger market than proprietary products could foster, which means more potential sales and revenue.

Standards allow companies to invest more strategically in R&D rather than inventing everything from scratch. Once common form factors are set, companies can base their designs on standards and focus on innovation.