ECE/CS 250
Computer Architecture

Course review

Tyler Bletsch

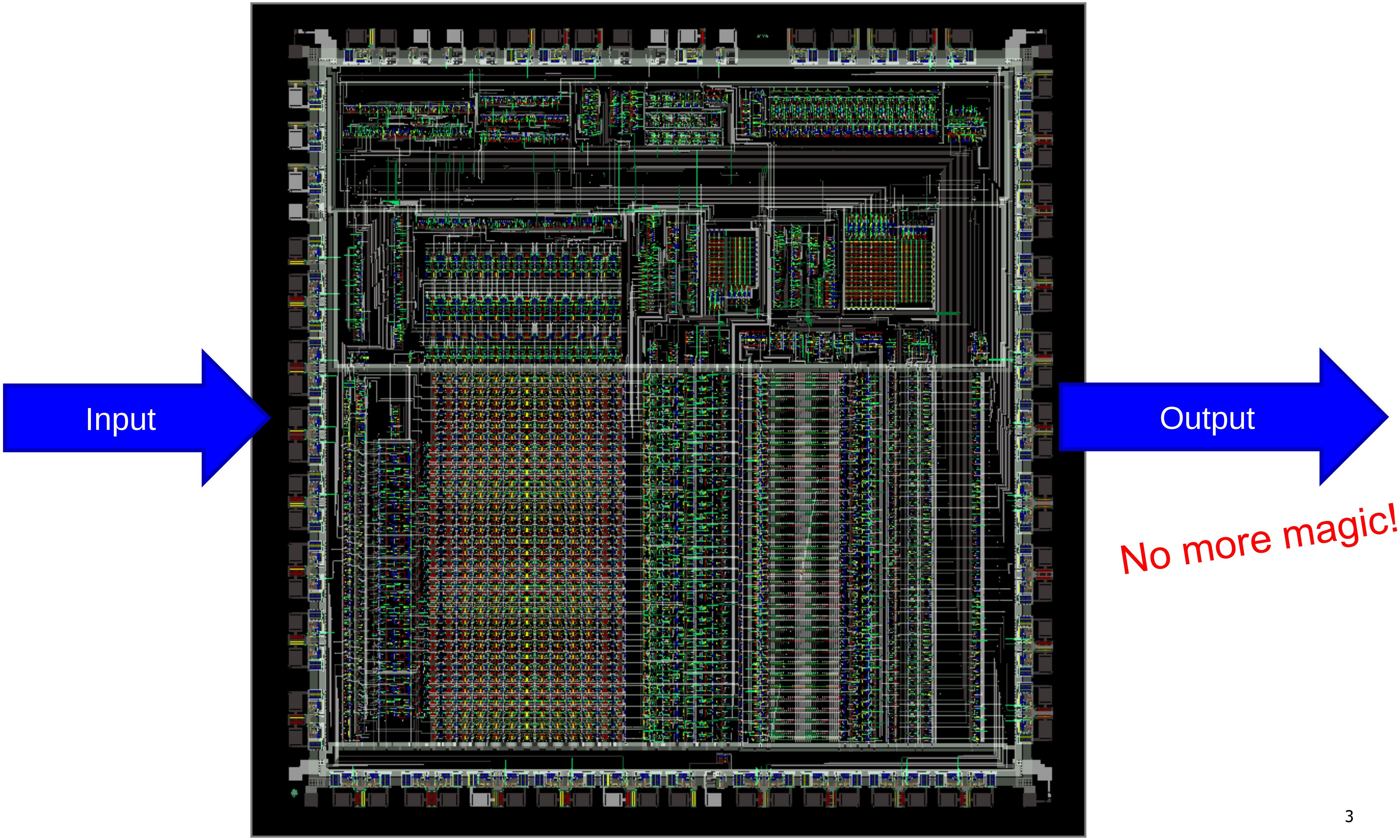
Duke University

Includes work by
Daniel J. Sorin (Duke), Amir Roth (Penn), and Alvin Lebeck (Duke)

Introduction

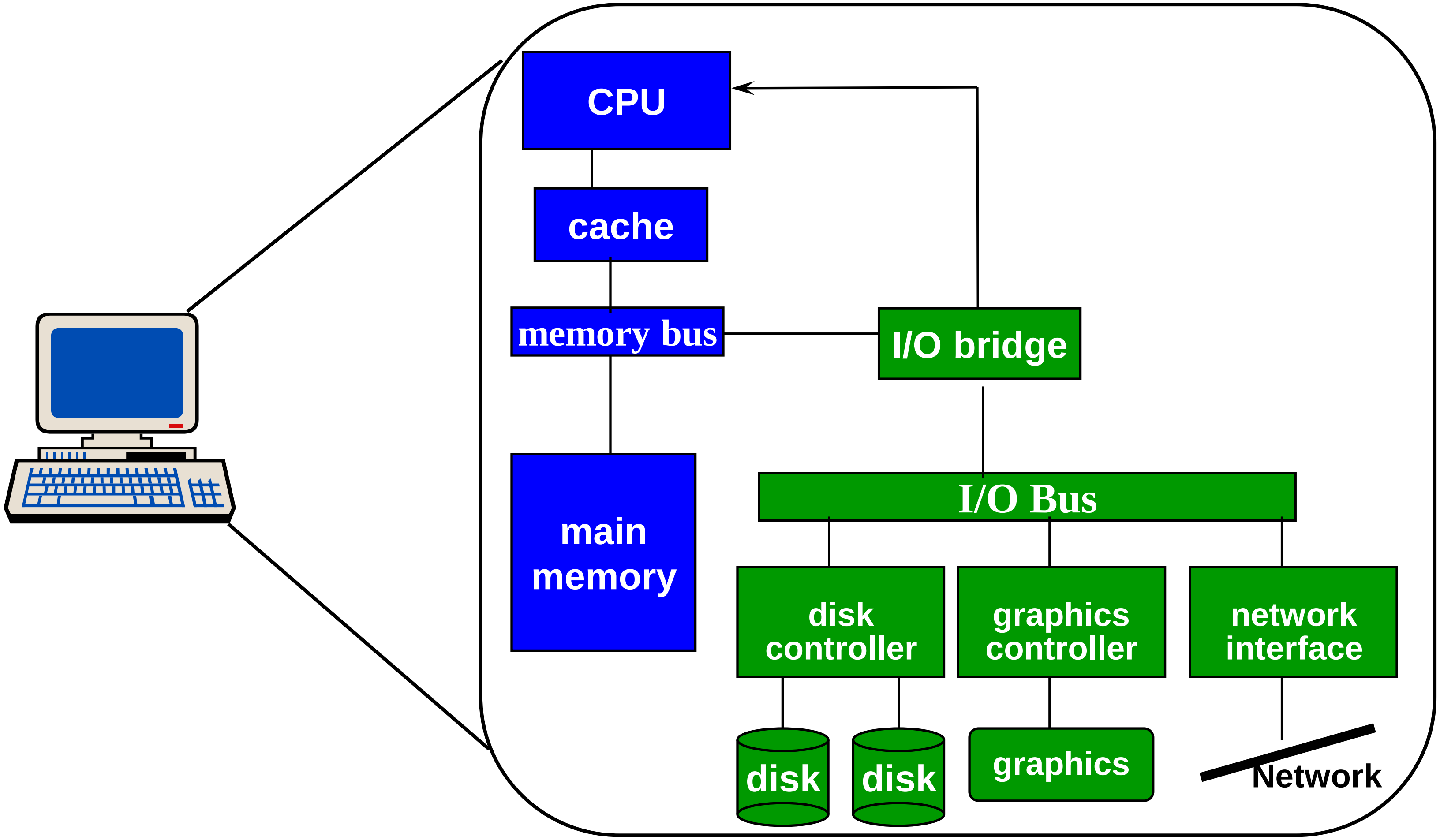
Course objective:
Evolve your understanding of computers

After



No more magic!

System Organization



C programming

What is C?

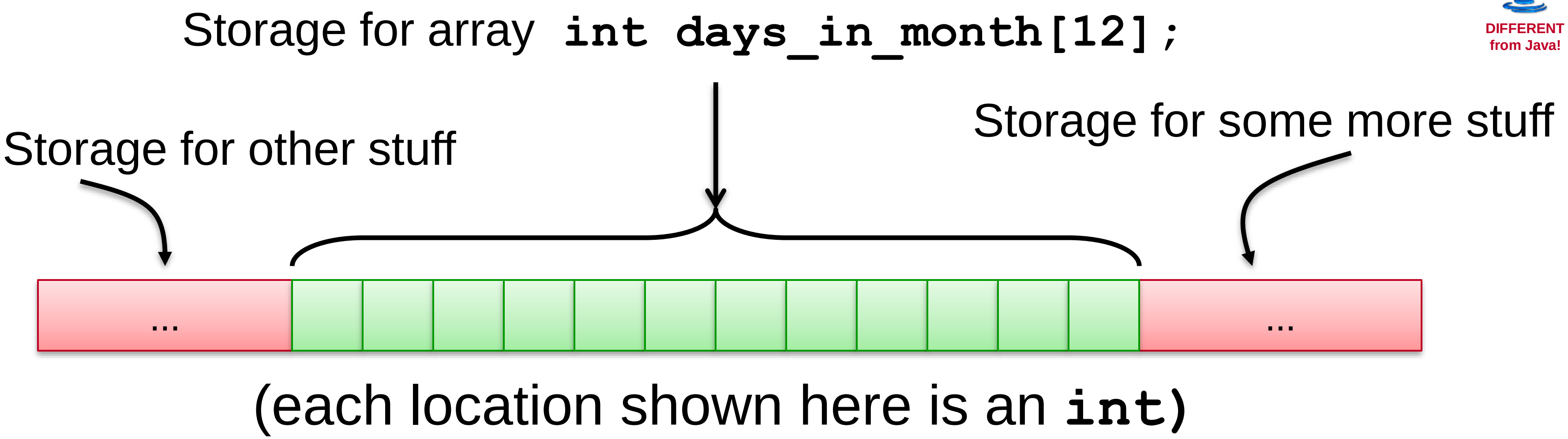
- The language of UNIX
- Procedural language (no classes)
- Low-level access to memory
- Easy to map to machine language
- Not much run-time stuff needed
- Surprisingly cross-platform

Why teach it now?
To expand from basic programming to
operating systems and embedded development.

Also, as a case study to understand computer architecture in general.

Memory Layout and Bounds Checking

DIFFERENT from Java!


Storage for array int days_in_month[12];
Storage for other stuff
Storage for some more stuff
…
…
(each location shown here is an int)

- There is NO bounds checking in C
  - i.e., it's legal (but not advisable) to refer to `days_in_month[216]` or `days_in_month[-35]` !
  - who knows what is stored there?

Structures

DIFFERENT from Java!

- Structures are sort of like Java objects
  - They have member variables
  - But they do NOT have methods!
- Structure definition with `struct` keyword

```
struct student_record {
    int id;
    float grade;
} rec1, rec2;
```

- Declare a variable of the structure type with `struct` keyword

```
struct student_record onerec;
```

- Access the structure member fields with dot (‘.’), e.g. `structvar.member`

```
onerec.id = 12;
onerec.grade = 79.3;
```

Let’s look at memory addresses!

- You can find the address of ANY variable with:

DIFFERENT from Java!

&

The address-of operator

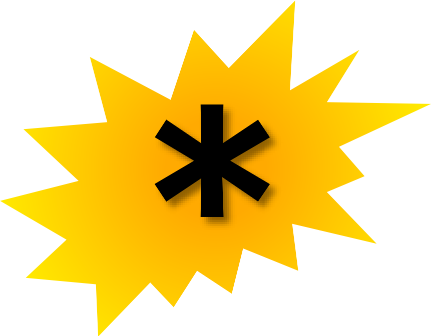
```
int v = 5;
printf(“%d\n”,v);
printf(“%p\n”,&v);
```

```
$ gcc x4.c && ./a.out
5
0x7fffd232228c
```

What's a pointer?

DIFFERENT from Java!

- It's a memory address you treat as a variable
- You declare pointers with:



The *dereference* operator

```
int v = 5;
int* p = &v;
printf(“%d\n”,v);
printf(“%p\n”,p);
```

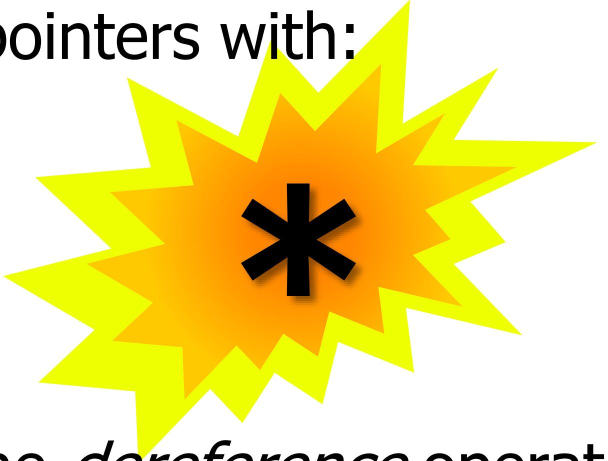
Append to any data type

```
$ gcc x4.c && ./a.out
5
0x7fffe0e60b7c
```

What’s a pointer?

DIFFERENT from Java!

- You can <u>look up</u> what’s stored *at* a pointer!
- You **dereference** pointers with:



The *dereference* operator

```
int v = 5;
int* p = &v;
printf(“%d\n”,v);
printf(“%p\n”,p);
printf(“%d\n”,*p);
```

Prepend to any pointer variable or expression

```
$ gcc x4.c && ./a.out
5
0x7fffe0e60b7c
5
```

C Memory Allocation

DIFFERENT from Java!

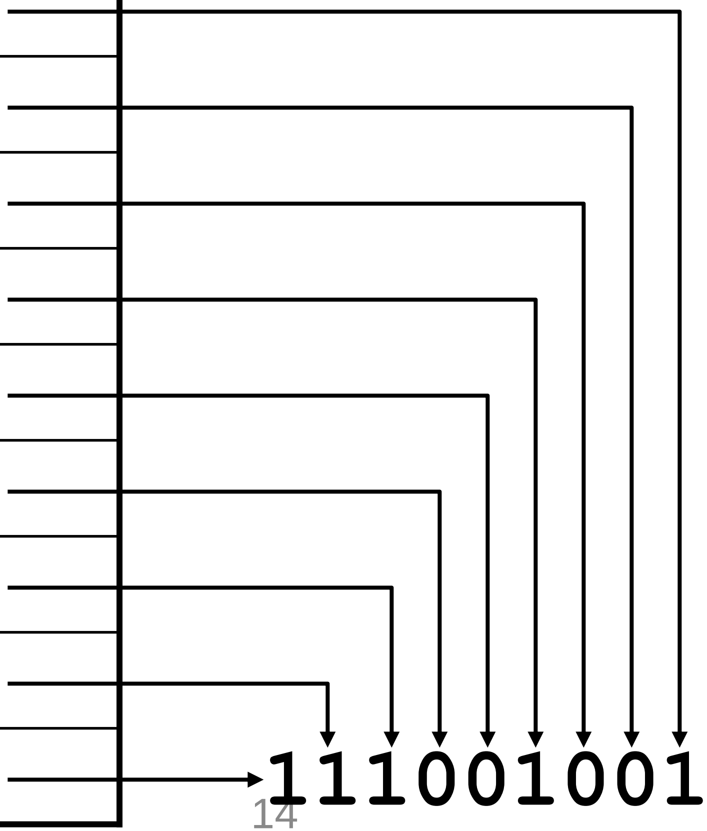
- **`void* malloc(nbytes)`**
  - Obtain storage for your data (like `new` in Java)
  - Often use `sizeof(type)` built-in returns bytes needed for `type`
  - `int* my_ptr = malloc (64);  // 64 bytes = 16 ints`
  - `int* my_ptr = malloc (64*sizeof(int)); // 64 ints`

- **`free(ptr)`**
  - Return the storage when you are finished (no Java equivalent)
  - `ptr` must be a value previously returned from malloc

Data representations and memory

Decimal to binary using remainders

?	Quotient	Remainder
457 ÷ 2 =	228	1
228 ÷ 2 =	114	0
114 ÷ 2 =	57	0
57 ÷ 2 =	28	1
28 ÷ 2 =	14	0
14 ÷ 2 =	7	0
7 ÷ 2 =	3	1
3 ÷ 2 =	1	1
1 ÷ 2 =	0	1



111001001

Decimal to binary using comparison

111001001

Num	Compare 2^n	≥ ?
457	256	1
201	128	1
73	64	1
9	32	0
9	16	0
9	8	1
1	4	0
1	2	0
1	1	1

Binary to/from hexadecimal

- 0101101100100011_2 -->
- 0101 1011 0010 0011_2 -->
- 5 B 2 3_{16}

1 F 4 B_{16} -->

0001 1111 0100 1011_2 -->

0001111101001011_2

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

2’s Complement Integers

- Use large positives to represent negatives
- $(-x) = 2^n - x$
- This is 1’s complement + 1
- $(-x) = 2^n - 1 - x + 1$
- **So, just invert bits and add 1**

6-bit examples:

$010110_2 = 22_{10}$; $101010_2 = -22_{10}$

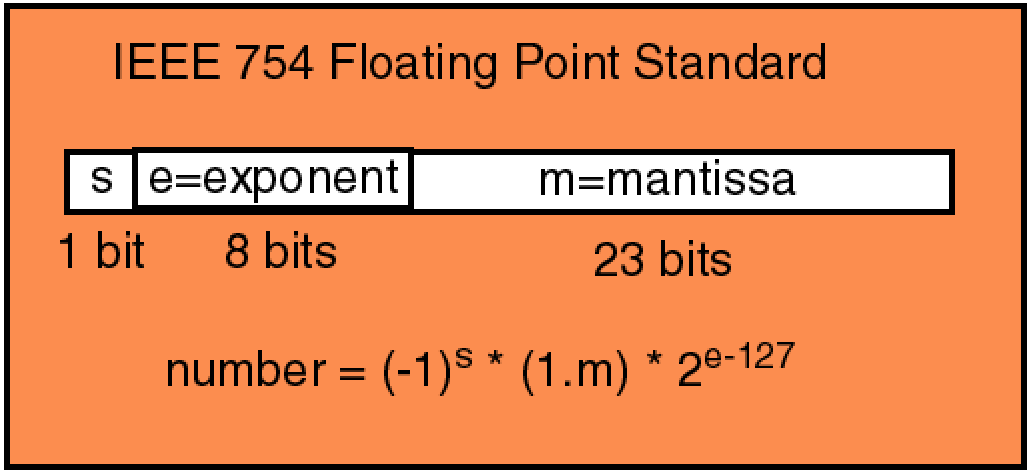
$1_{10} = 000001_2$; $-1_{10} = 111111_2$

$0_{10} = 000000_2$; $-0_{10} = 000000_2$ → good!

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Floating point

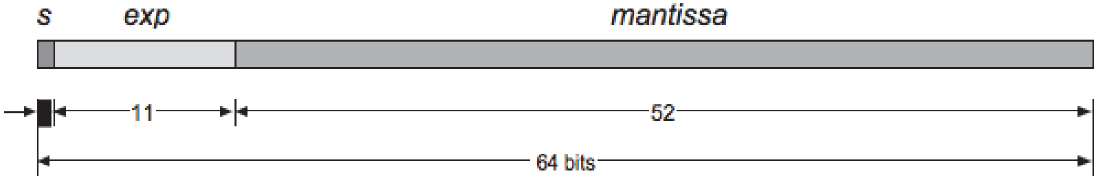
- 32-bit **float** format:



IEEE 754 Floating Point Standard

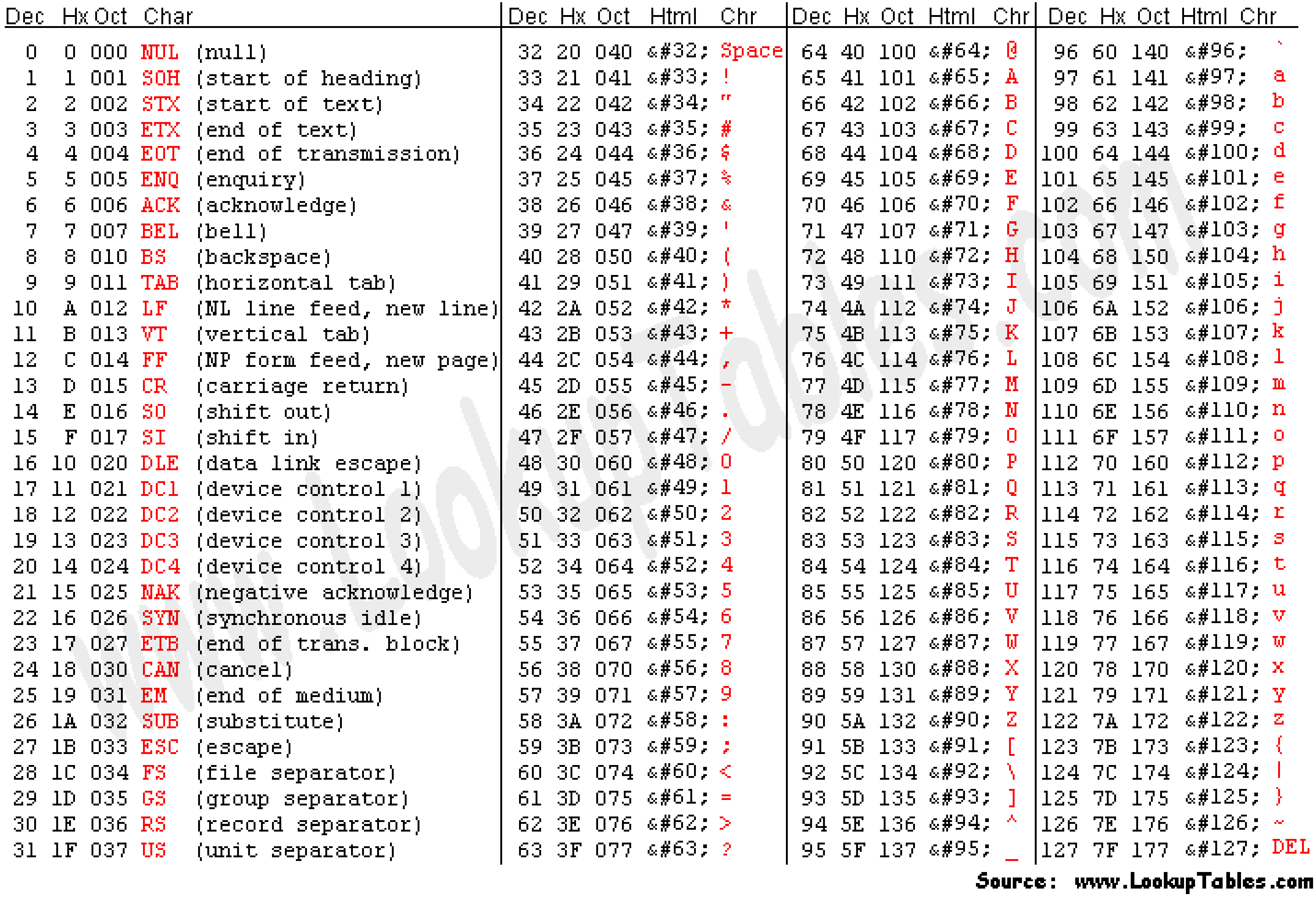
number = $(-1)^{s} * (1.m) * 2^{e-127}$

- 64-bit **double** format:
  (same thing, but with more bits)



Double Precision

Standardized ASCII (0-127)

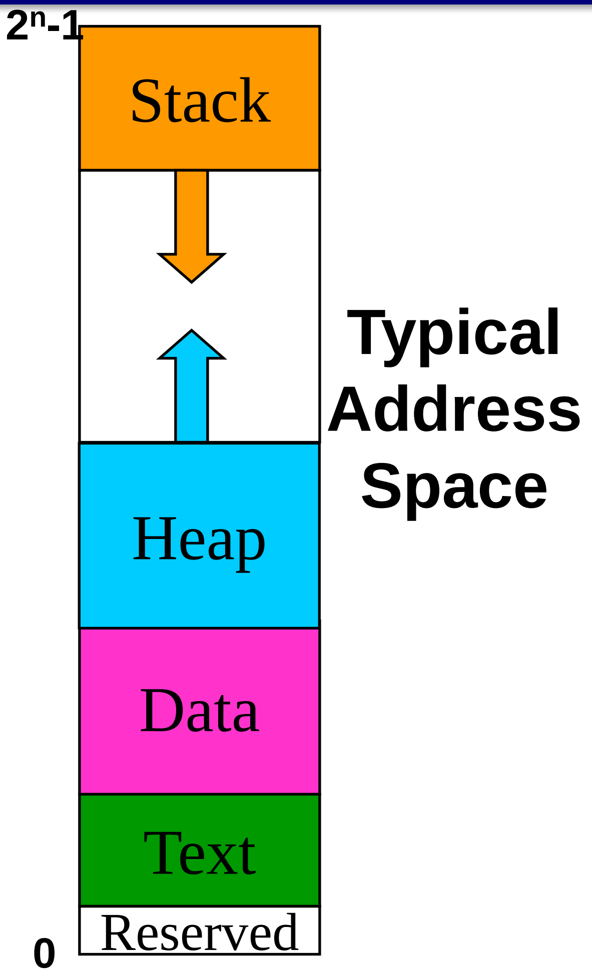


Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Memory Layout

- Memory is array of bytes, but there are conventions as to what goes where in this array
- Text: instructions (the program to execute)
- Data: global variables
- Stack: local variables and other per-function state; starts at top & grows down
- Heap: dynamically allocated variables; grows up
- What if stack and heap overlap????



Learning Assembly language with MIPS

The MIPS architecture

- 32-bit word size
- 32 registers ($0 is zero, $31 is return address)
- Fixed size 32-bit aligned instructions
- Types of instructions:
  - Math and logic:
    - `or $1, $2, $3` → $1 = $2 | $3
    - `add $1, $2, $3` → $1 = $2 + $3
  - Loading constants:
    - `li $1, 50` → $1 = 50
  - Memory:
    - `lw $1, 4($2)` → $1 = *($2 + 4)
    - `sw $1, 4($2)` → *($2 + 4) = $1
  - Control flow:
    - `j label` → PC = label
    - `bne $1, $2, label` → if ($1==$2) PC=label

Control Idiom: If-Then-Else

- Control idiom: **if-then-else**

```
if (A < B) A++;       // assume A in register $1
else B++;             // assume B in $2
```

```
       slt  $3,$1,$2        // if $1<$2, then $3=1
       beqz $3,else        // branch to else if !condition
       addi $1,$1,1
       j    join            // jump to join
else:  addi $2,$2,1
join:
```

ICQ: assembler converts "else" operand of beqz into immediate → what is the immediate?

MIPS Register Usage/Naming Conventions

0	zero constant	
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		
15	t7	

16	s0	callee saves
. . .		
23	s7	
24	t8	temporary (cont’d)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Also 32 floating-point registers: $f0 .. $f31

Important: The only general purpose registers are the $s and $t registers.

Everything else has a specific usage:
$a = arguments, $v = return values, $ra = return address, etc.

MIPS Instruction Formats

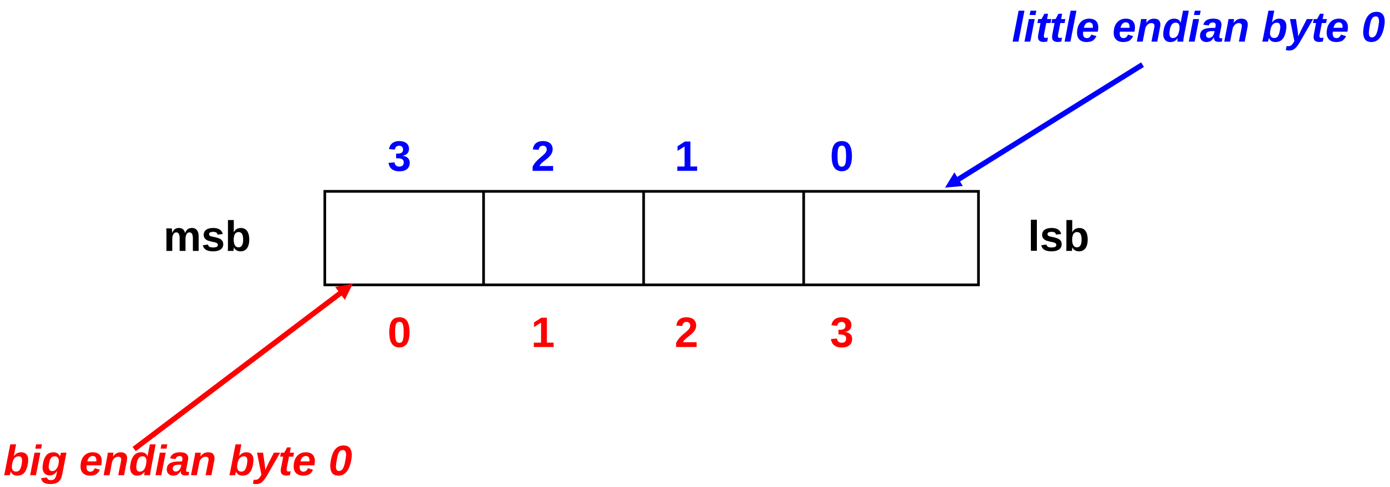
- 3 variations on theme from previous slide
  - All MIPS instructions are either R, I, or J type
  - Note: all instructions have opcode as first 6 bits

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)
I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)		
J-type	Op(6)	Target(26)				

Memory Addressing Issue: Endian-ness

Byte Order

- Big Endian: byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, SPARC, HP PA-RISC
- Little Endian: byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC/Compaq Alpha



Combinational logic

Truth Tables

- Map any number if inputs to any number of outputs
- Example:

(A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

Compute Output

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Convert truth table to function

- Given a Truth Table, find the formula?

Write down every “true” case

Then OR together:

```
(!A & !B & !C)  |
(!A & !B & C)   |
(!A & B & !C)  |
(A & B &!C)  |
(A & B &C)
```

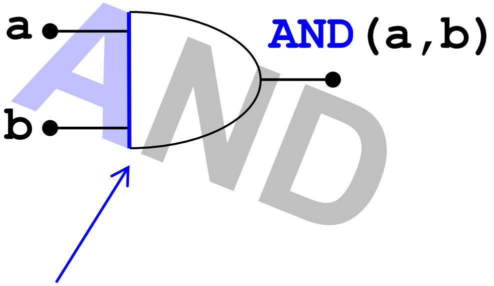
A	B	C	Output
0	**0**	**0**	**1**
0	**0**	**1**	**1**
0	**1**	**0**	**1**
0	1	1	0
1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	**1**	**1**	**1**

Summary of all Boolean axioms

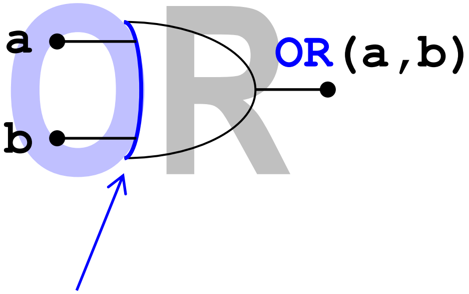
Name	AND form	OR form
Identity law	1 & A = A	0 | A = A
Null law	0 & A = 0	1 | A = 1
Idempotent law	A & A = A	A | A = A
Inverse law	A & !A = 0	A | !A = 1
Commutative law	A & B = B & A	A | B = B | A
Associative law	(A&B) & C = A & (B&C)	(A|B) | C = A | (B|C)
Distributive law	A | (B&C) = (A|B) & (A|C)	A & (B|C) = (A&B) | (A&C)
Absorption law	A & (A|B) = A	A | (A&B) = A
De Morgan's law	!(A&B) = !A | !B	!(A|B) = !A & !B
Double negation law	!!A = A	

Adapted from http://studytronics.weebly.com/boolean-algebra.html

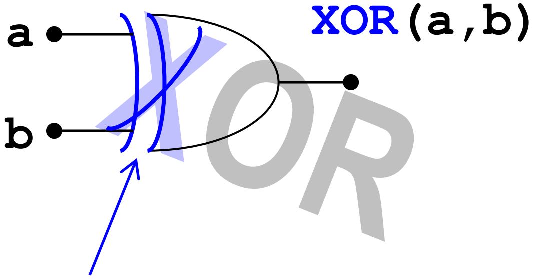
Guide to Remembering your Gates



Straight like an A

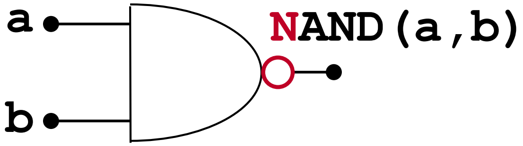


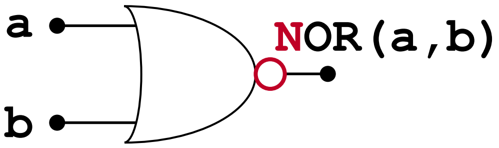
Curved, like an O

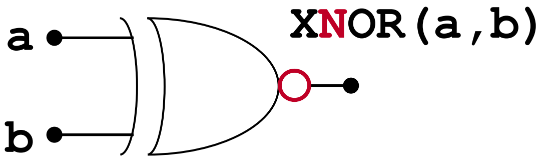


XOR looks like OR (curved line), but has two lines (like an X does)

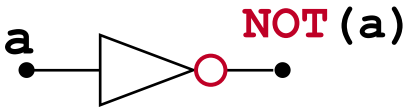
Circle means NOT







(XNOR is 1-bit “equals” by the way)



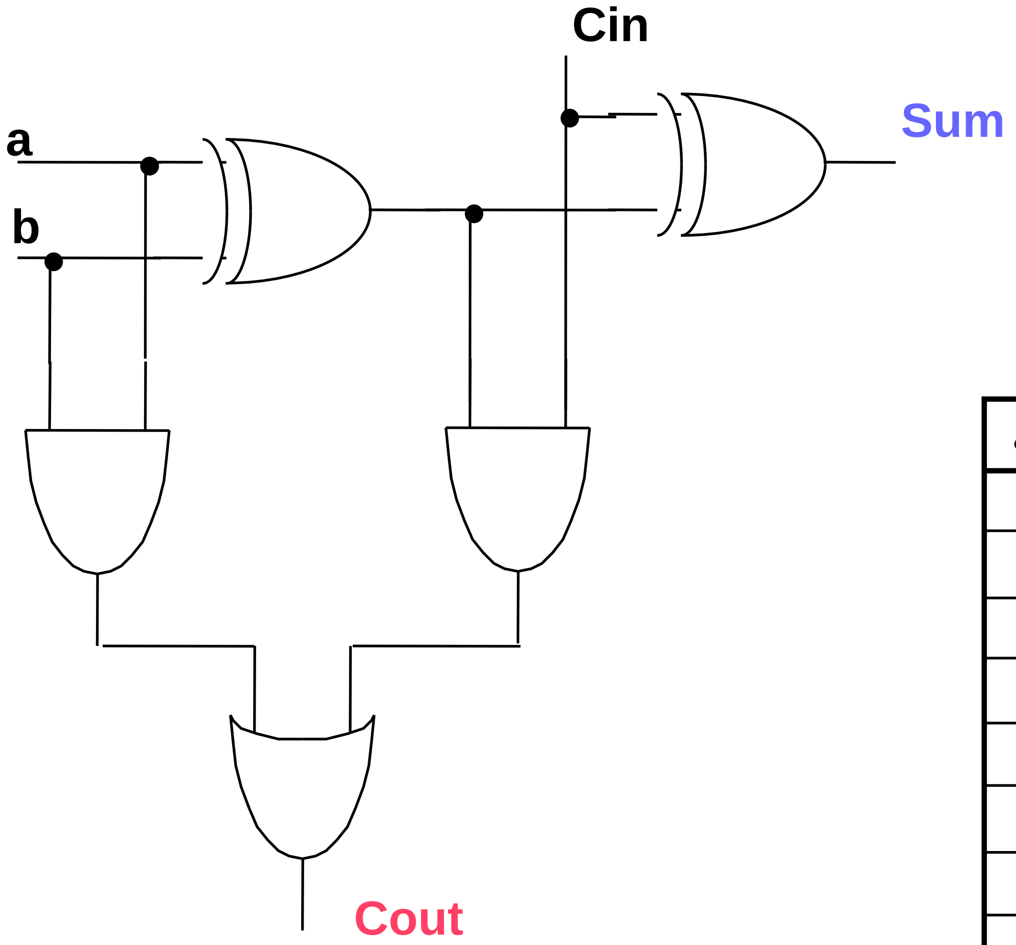
Designing a 1-bit adder

- So we’ll need to add three bits (including carry-in)
- Two-bit output is the **carry-out** and the **sum**

a		b		C_{in}		
0	+	0	+	0	=	00
0	+	0	+	1	=	01
0	+	1	+	0	=	01
0	+	1	+	1	=	10
1	+	0	+	0	=	01
1	+	0	+	1	=	10
1	+	1	+	0	=	10
1	+	1	+	1	=	11

Turn into expression,
simplify,
circuit-ify,
yadda yadda yadda…

A 1-bit Full Adder

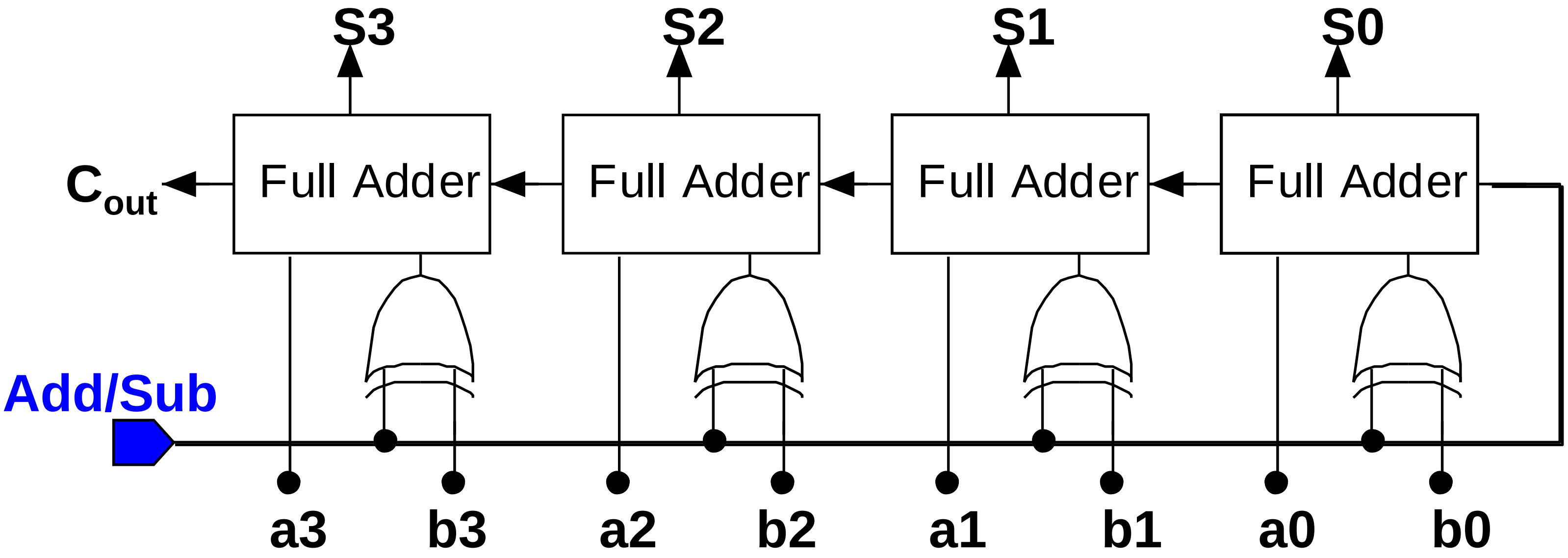


```
 01101100
 01101101
+00101100
 10011001
```

a	b	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

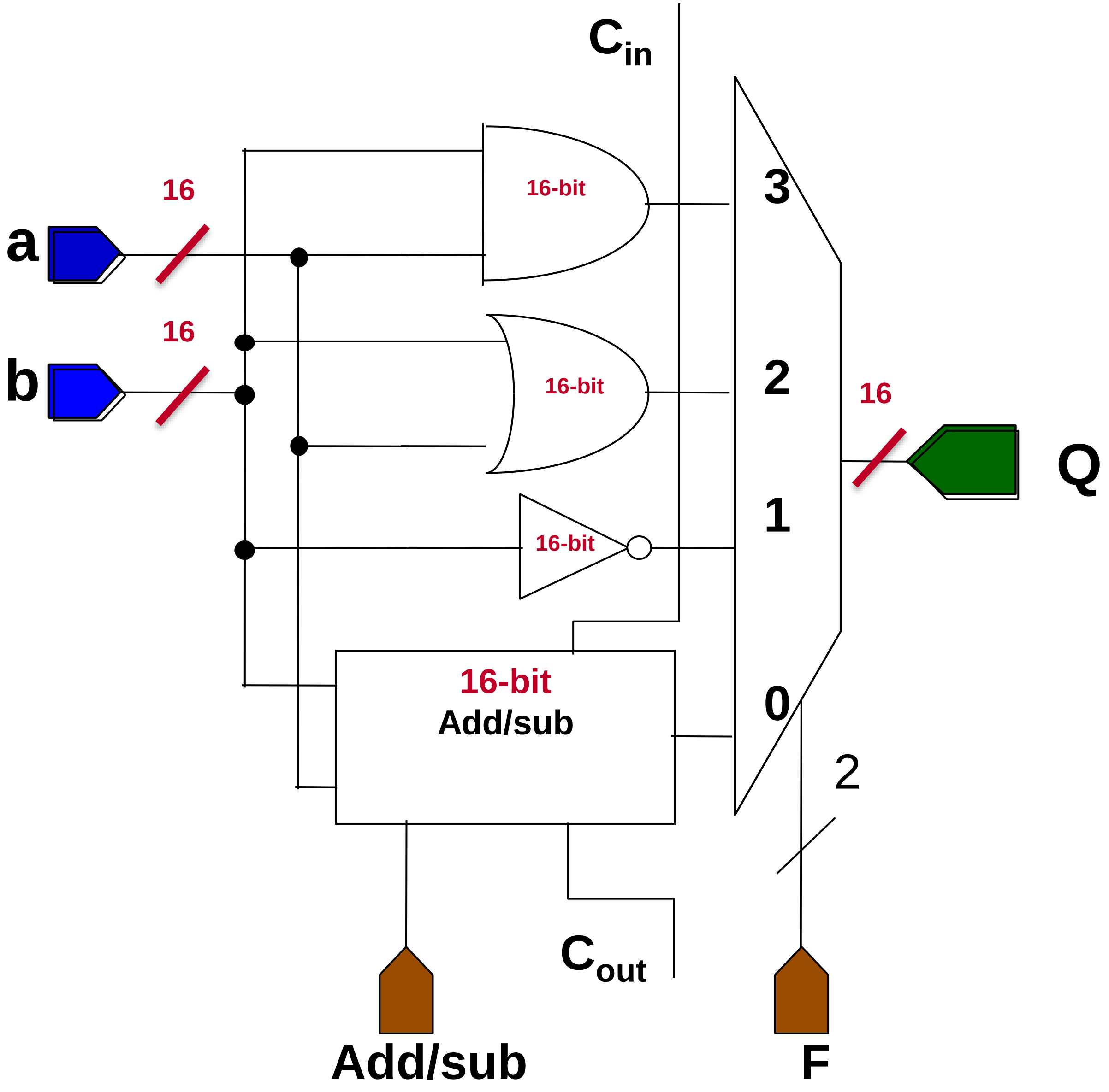
Logisim example
basic_logic.circ : full-adder

Example: Adder/Subtractor



Logisim example
basic_logic.circ : 4bit-addsub

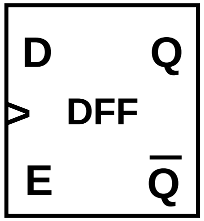
The ALU

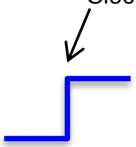

C_in
16
a
16-bit
3
16
b
16-bit
2
16
Q
1
16-bit
16-bit
Add/sub
0
2
C_out
Add/sub
F

Sequential logic

D flip flops

- Stores one bit
- Inputs:
  - The data D
  - The clock '>'
  - An "enable" signal E
- Outputs:
  - The stored bit output Q (and also its inverse !Q)
- "Commits" the input bit on **clock *rise***, and only if E is high

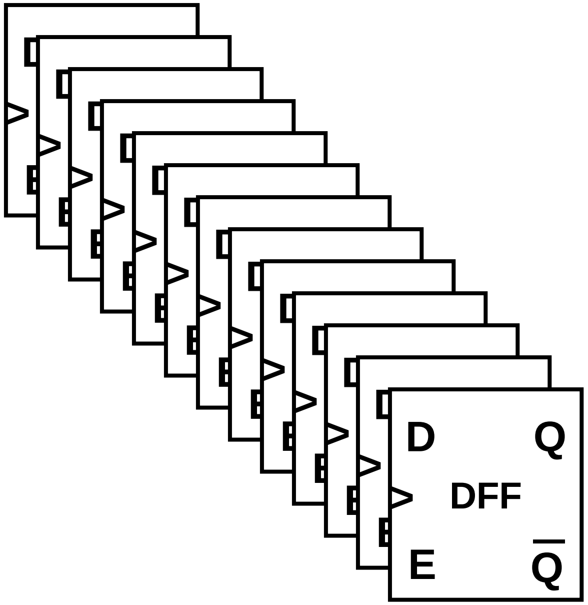




Clock rise (bit gets saved at this time)

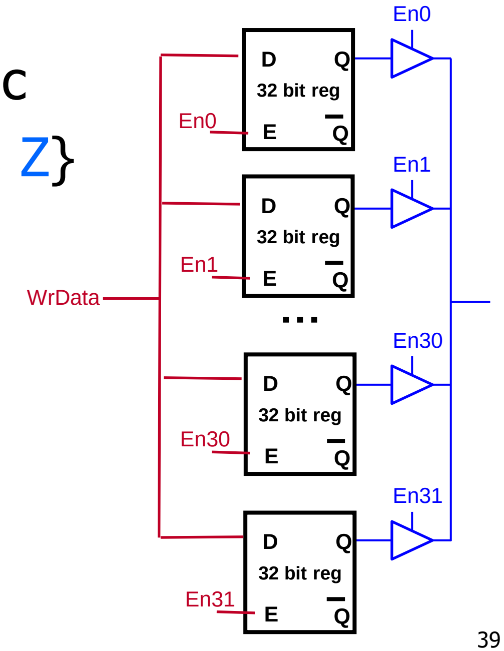
Register

- **Register**: N flip flops working in parallel, where N is the word size



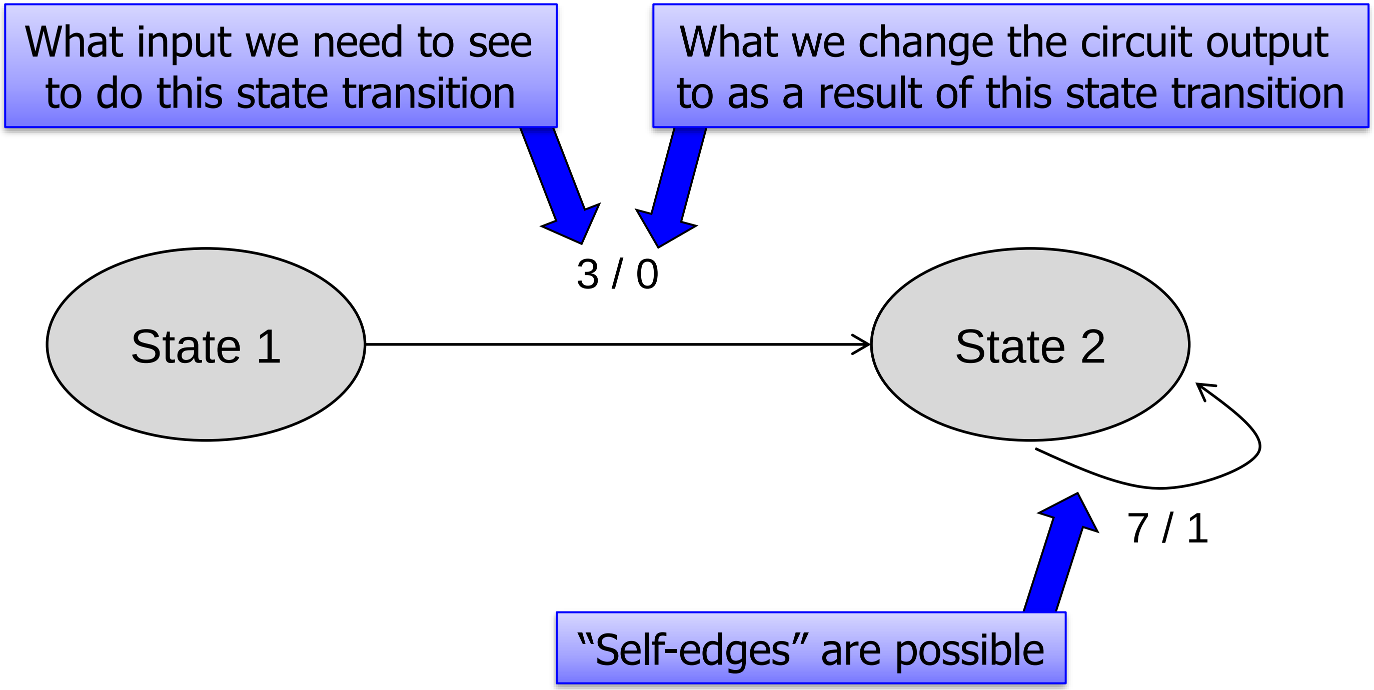
Register file

- A set of registers with multiple ports so numbered registers can be read/written.
- How to **write**:
  - Use decoder to convert reg # to one hot
  - Send write data to all regs
  - Use one hot encoding of reg # to enable right reg
- How to **read**:
  - 32 input mux (the way we've made it) not realistic
  - To do this: expand our world from {1,0} to {1, 0, Z}



Finite state machines

How FSMs are represented



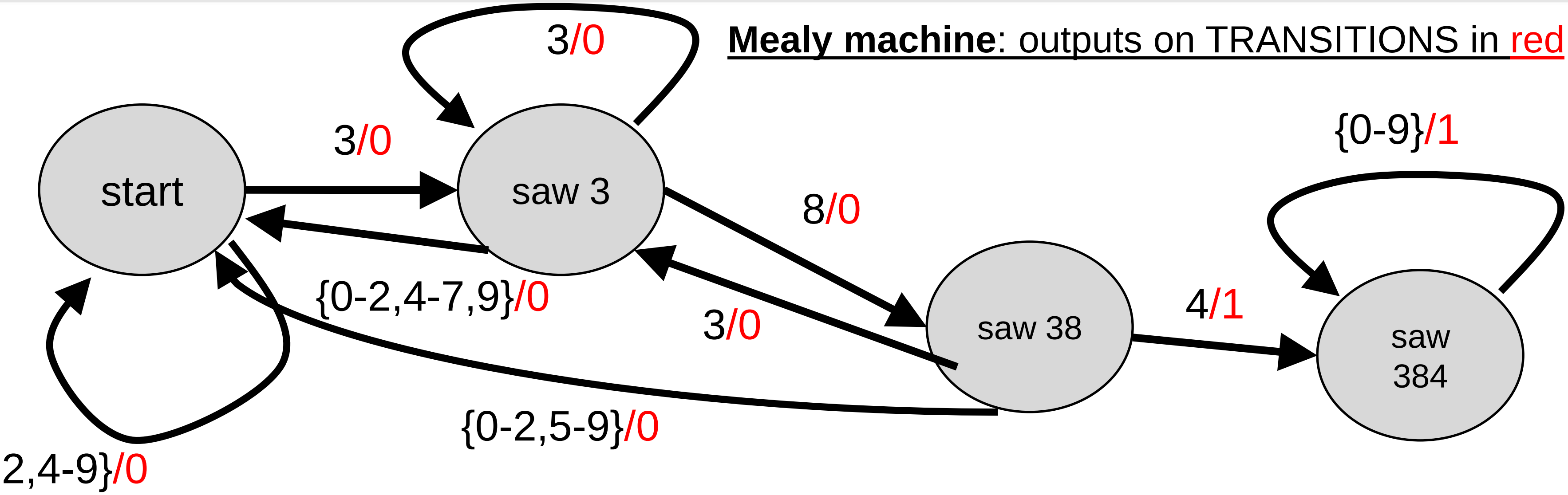
What input we need to see to do this state transition

What we change the circuit output to as a result of this state transition

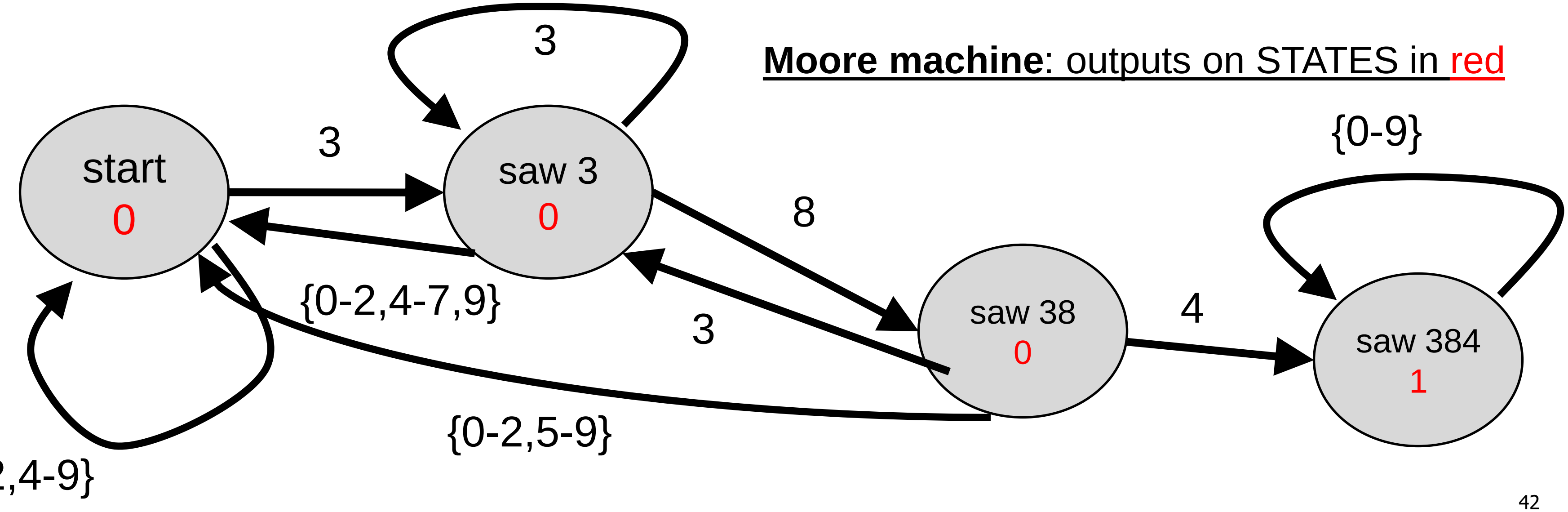
“Self-edges” are possible

Mealy vs Moore

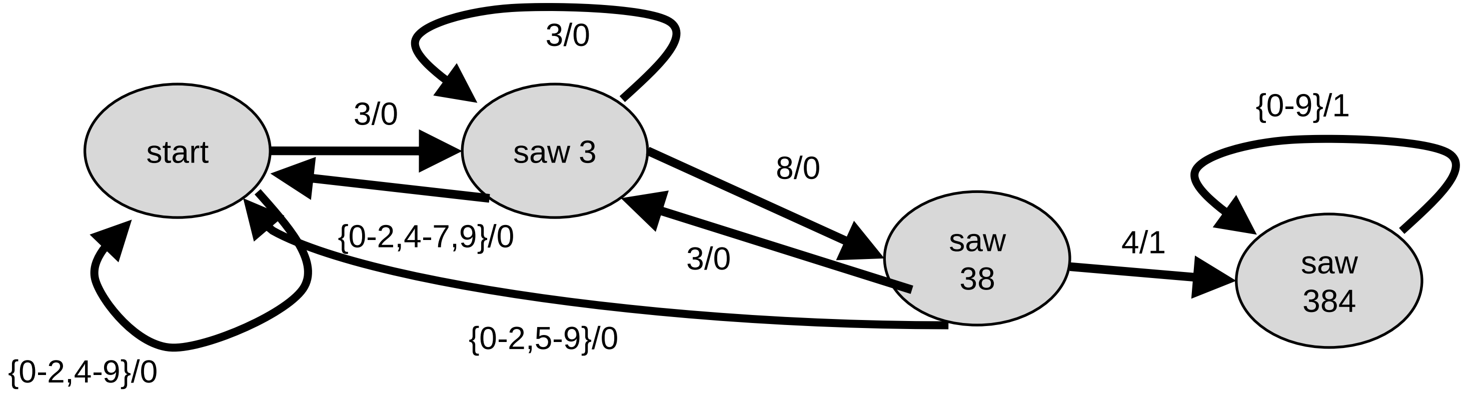
Mealy machine: outputs on TRANSITIONS in red



Moore machine: outputs on STATES in red



State Transition Diagram → Truth Table



Current State	Input	Next state	Output
Start	3	Saw 3	0 (closed)
Start	Not 3	Start	0
Saw 3	8	Saw 38	0
Saw 3	3	Saw 3	0
Saw 3	Not 8 or 3	Start	0
Saw 38	4	Saw 384	1 (open)
Saw 38	3	Saw 3	0
Saw 38	Not 4 or 3	Start	0
Saw 384	Any	Saw 384	1

State Transition Diagram → Truth Table

Current State	Input	Next state	Output
00 (start)	3	01	0 (closed)
00	Not 3	00	0
01	8	10	0
01	3	01	0
01	Not 8 or 3	00	0
10	4	11	1 (open)
10	3	01	0
10	Not 4 or 3	00	0
11	Any	11	1

4 states → 2 flip-flops to hold the current state of the FSM
inputs to flip-flops are D_1D_0
outputs of flip-flops are Q_1Q_0

State Transition Diagram → Truth Table

Q1	Q0	Input	D1	D0	Output
0	0	3	0	1	0 (closed)
0	0	Not 3	0	0	0
0	1	8	1	0	0
0	1	3	0	1	0
0	1	Not 8 or 3	0	0	0
1	0	4	1	1	1 (open)
1	0	3	0	1	0
1	0	Not 4 or 3	0	0	0
1	1	Any	1	1	1

Input can be 0-9 → requires 4 bits
input bits are in3, in2, in1, in0

State Transition Diagram → Truth Table

Q1	Q0	In3	In2	In1	In0	D1	D0	Output
0	0	0	0	1	1	0	1	0
0	0	Not 3 (all binary combos other than 00011)				0	0	0
0	1	1	0	0	0	1	0	0
0	1	0	0	1	1	0	1	0
0	1	Not 8 or 3 (all binary combos other than 01000 & 00011)				0	0	0
1	0	0	1	0	0	1	1	1
1	0	0	0	1	1	0	1	0
1	0	Not 4 or 3 (all binary combos other than 00100 & 00011)				0	0	0
1	1	Any				1	1	1

From here, it’s just like combinational logic design!
Write out product-of-sums equations, optimize, and build.

State Transition Diagram → Truth Table

Q1	Q0	In3	In2	In1	In0	D1	D0	Output
0	0	0	0	1	1	0	1	0
0	0	Not 3				0	0	0
0	1	1	0	0	0	1	0	0
0	1	0	0	1	1	0	1	0
0	1	Not 8 or 3				0	0	0
1	0	0	1	0	0	1	1	1
1	0	0	0	1	1	0	1	0
1	0	Not 4 or 3				0	0	0
1	1	Any				1	1	1

Output = (Q1 & !Q0 & !In3 & In2 & !In1 & !In0) | (Q1 & Q0)

D1 = (!Q1 & Q0 & In3 & !In2 & !In1 & !In0) | (Q1 & !Q0 & !In3 & In2 & !In1 & !In0) | (Q1 & Q0)

D0 = do the same thing

State Transition Diagram → Truth Table

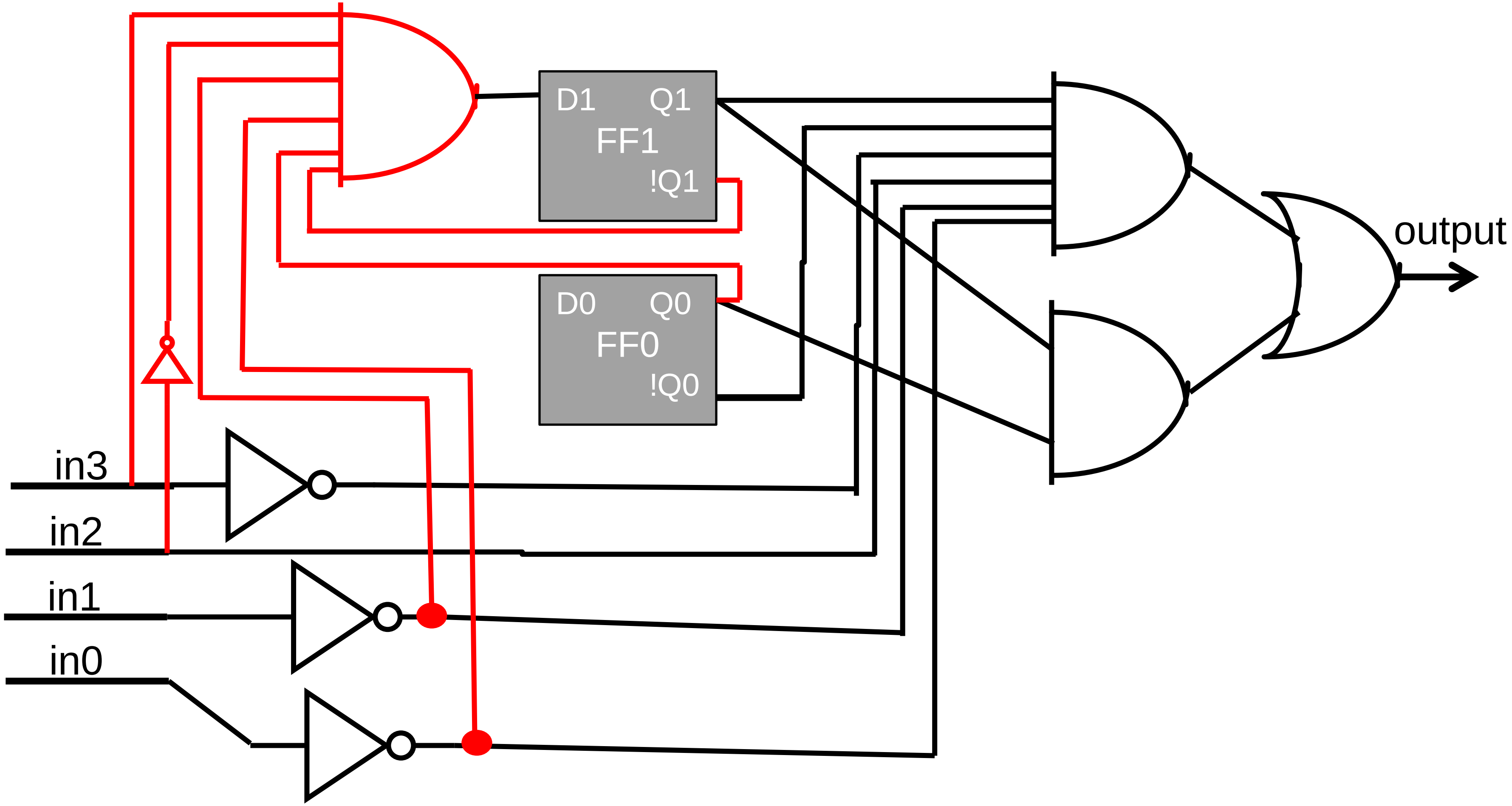
Q1	Q0	In3	In2	In1	In0	D1	D0	Output
0	0	0	0	1	1	0	1	0
0	0	Not 3				0	0	0
0	1	1	0	0	0	1	0	0
0	1	0	0	1	1	0	1	0
0	1	Not 8 or 3				0	0	0
1	0	0	1	0	0	1	1	1
1	0	0	0	1	1	0	1	0
1	0	Not 4 or 3				0	0	0
1	1	Any				1	1	1

Remember, these represent **DFF outputs**

…and these are the **DFF inputs**

The DFFs are how we store the **state**.

Truth Table → Sequential Circuit

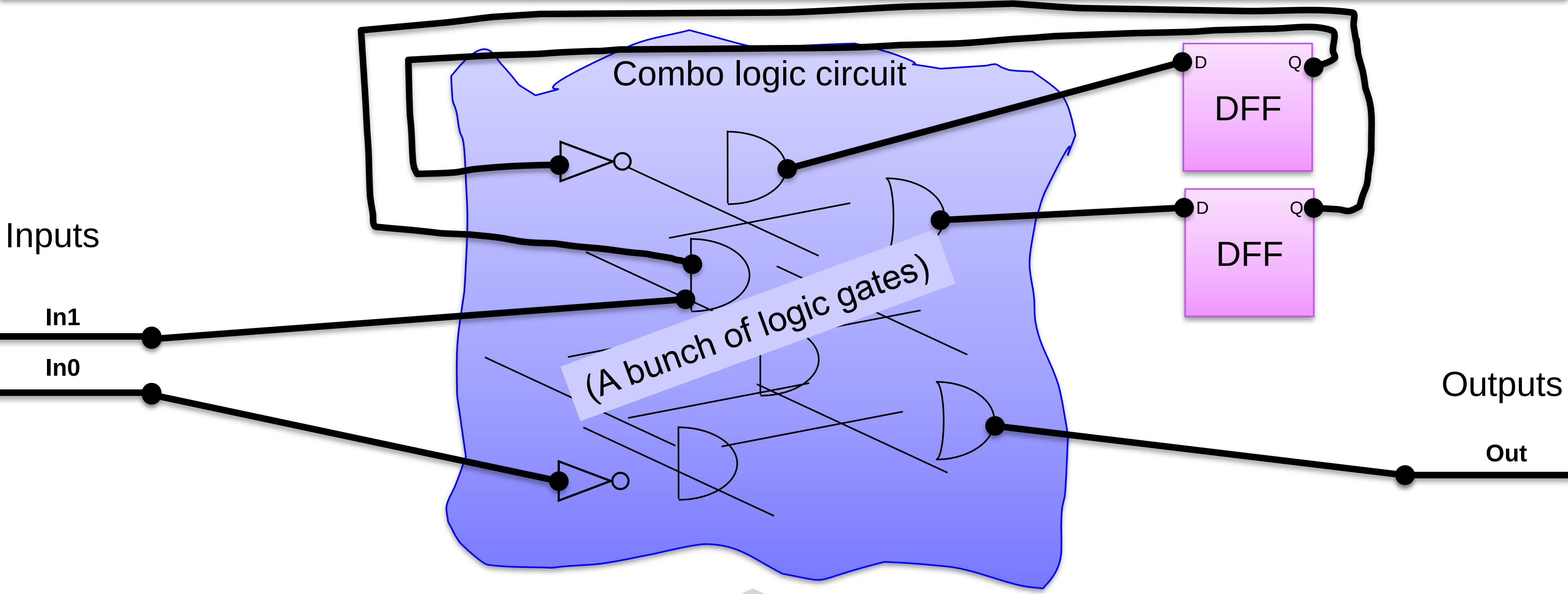


D1 = (!Q1 & Q0 & In3 & !In2 & !In1 & !In0) | (Q1 & !Q0 & !In3 & In2 & !In1 & !In0) | (Q1 & Q0)

Not pictured

Follow a similar procedure for D0…

How to think about the FSM circuit



Yields

Truth table

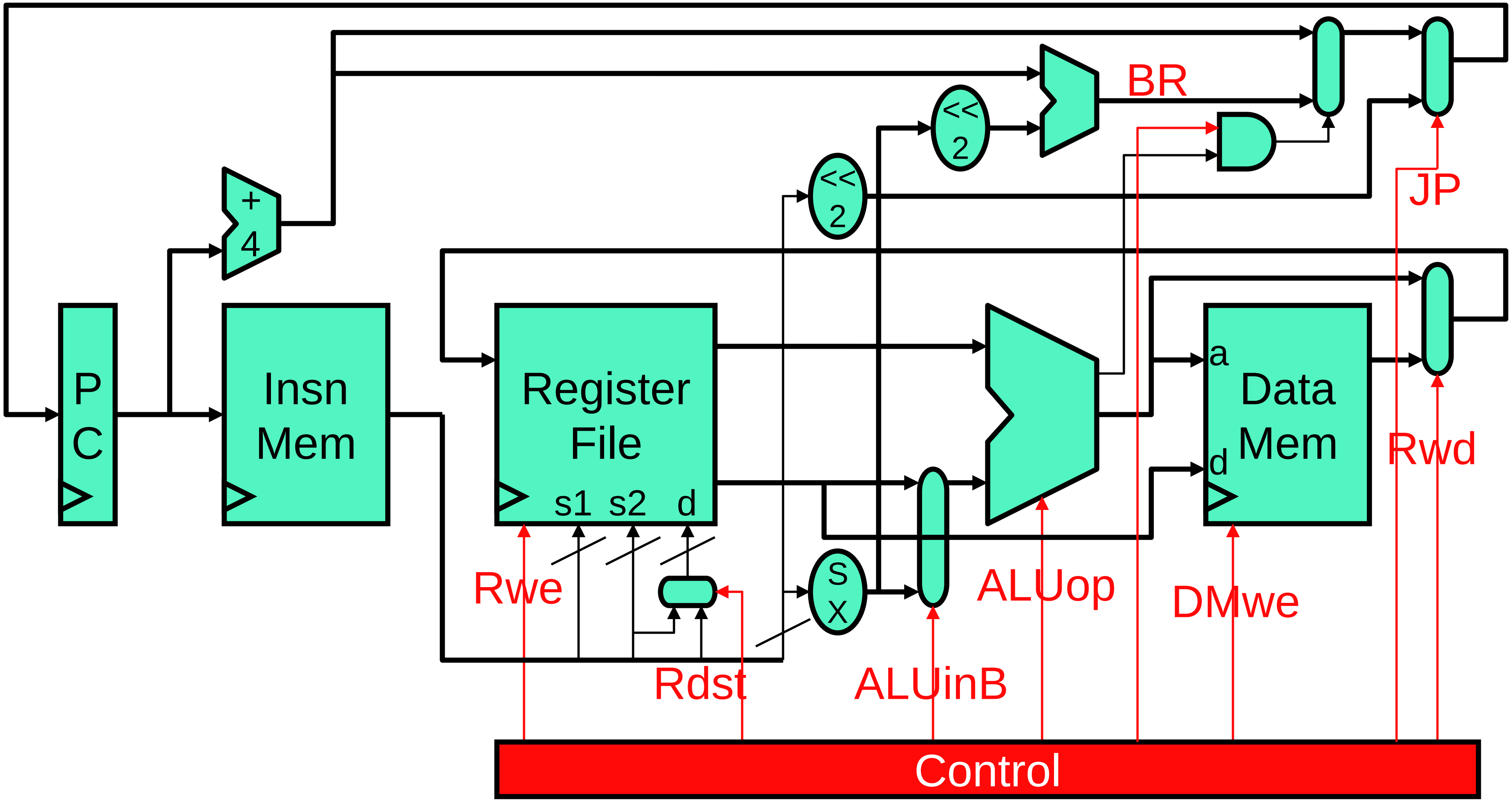
Current state		Input		Next state		Output
Q1	Q0	In1	In0	D1	D0	Out
0	0	0	0	0	1	1
0	0	0	1	1	0	1
0	0	1	0	1	1	0
0	0	1	1	0	1	0

Steps:

1. Do truth table
2. Convert to logic circuit
3. Slap down DFFs
4. Hook up DFFs
5. Hook up inputs/outputs

CPU datapath and control

The overall datapath

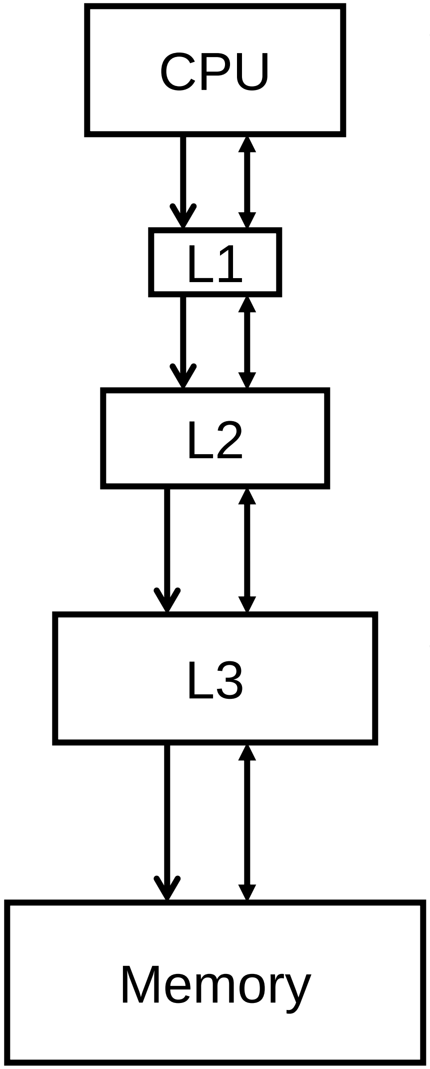


Exceptions

- **Exceptions and interrupts**
  - Infrequent (exceptional!) events
    - I/O, divide-by-0, illegal instruction, page fault, protection fault, ctrl-C, ctrl-Z, timer
  - Handling requires intervention from operating system
    - End program: divide-by-0, protection fault, illegal insn, ^C
    - Fix and restart program: I/O, page fault, ^Z, timer
  - Handling should be transparent to application code
    - Don't want to (can't) constantly check for these using insns
    - Want "Fix and restart" equivalent to "never happened"

Caching

Big Concept: Memory Hierarchy



- Use hierarchy of memory components
  - Upper components (closer to CPU)
    - Fast ↔ Small ↔ Expensive
  - Lower components (further from CPU)
    - Slow ↔ Big ↔ Cheap
  - Bottom component (for now!) = what we have been calling "memory" until now

- Make average access time close to L1's
  - How?
  - Most frequently accessed data in L1
  - L1 + next most frequently accessed in L2, etc.
  - **Automatically** move data up&down hierarchy

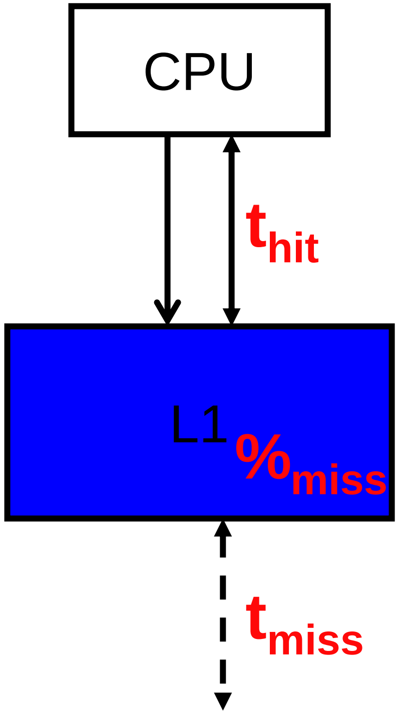
Terminology

- **Hit**: Access a level of memory and find what we want
- **Miss**: Access a level of memory and DON'T find what we want

- **Block**: a group of spatially contiguous and aligned bytes

- **Temporal locality**: Recently accessed stuff likely to be accessed again soon
- **Spatial locality**: Stuff near recently accessed thing likely to be accessed soon

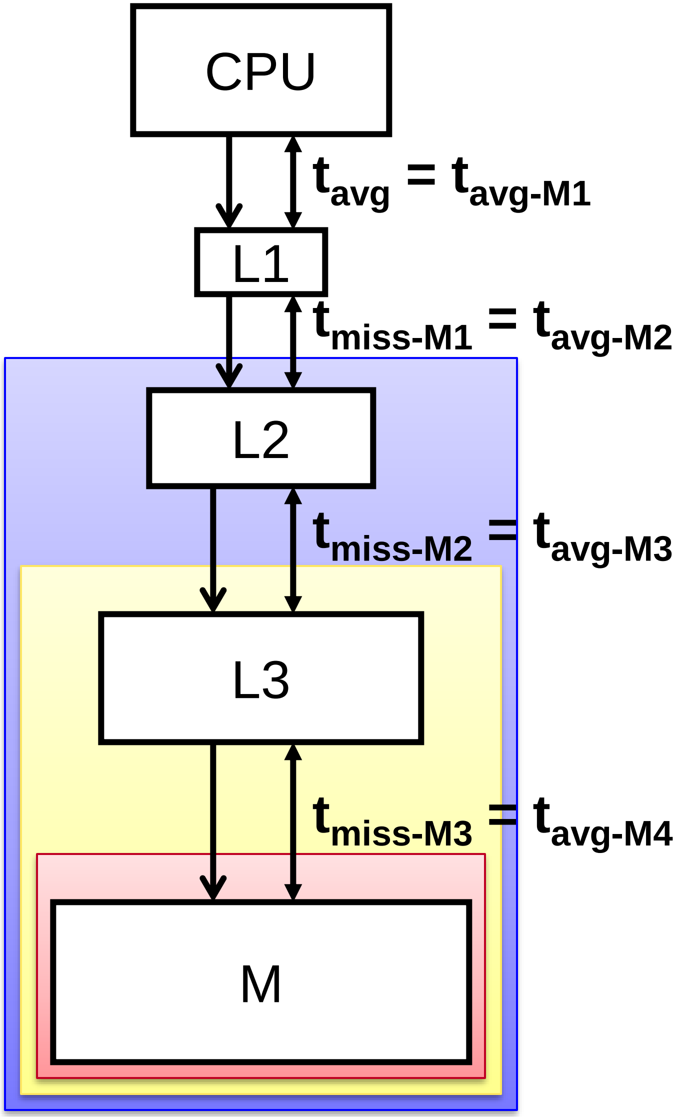
Memory Performance Equation



- For memory component L1
  - **Access**: read or write to L1
  - **Hit**: desired data found in L1
  - **Miss**: desired data not found in L1
    - Must get from another (slower) component
  - **Fill**: action of placing data in L1
  - $\%_{miss}$ (miss-rate): #misses / #accesses
  - t_{hit}: time to read data from (write data to) L1
  - t_{miss}: time to read data into M from lower level
- Performance metric
  - t_{avg}: average access time

$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

Abstract Hierarchy Performance


CPU
$t_{avg} = t_{avg\text{-}M1}$
L1
$t_{miss\text{-}M1} = t_{avg\text{-}M2}$
L2
$t_{miss\text{-}M2} = t_{avg\text{-}M3}$
L3
$t_{miss\text{-}M3} = t_{avg\text{-}M4}$
M

How do we compute t_{avg} ?

$=t_{avg\text{-}L1}$

$=t_{hit\text{-}L1} +(\%_{miss\text{-}L1} {}^{*}t_{miss\text{-}L1})$

$=t_{hit\text{-}L1} +(\%_{miss\text{-}L1} {}^{*}t_{avg\text{-}L2})$

$=t_{hit\text{-}L1} +(\%_{miss\text{-}L1} {}^{*}(t_{hit\text{-}L2}+(\%_{miss\text{-}L2} {}^{*}t_{miss\text{-}L2})))$

$=t_{hit\text{-}L1} +(\%_{miss\text{-}L1} {}^{*}(t_{hit\text{-}L2}+(\%_{miss\text{-}L2} {}^{*}t_{avg\text{-}L3})))$

= …

Note: Miss at level X = access at level X+1

Where to Put Blocks in Cache

- How to decide which frame holds which block?
  - And then how to find block we're looking for?
- Some more cache structure:
  - Divide cache into sets
    - A block can only go in its set → there is a 1-to-1 mapping from block address to set
  - Each set holds some number of frames = **set associativity**
    - E.g., 4 frames per set = 4-way set-associative
- At extremes
  - Whole cache has just one set = **fully associative**
    - Most flexible (longest access latency)
  - Each set has 1 frame = 1-way set-associative = "**direct mapped**"
    - Least flexible (shortest access latency)

Cache structure math

- Given capacity, block_size, ways (associativity), and word_size.
- Cache parameters:
  - num_frames = capacity / block_size
  - sets = num_frames / ways = capacity / block_size / ways
- Address bit fields:

Tag	Index	Block offset

  - offset_bits = $\log_2$(block_size)
  - index_bits = $\log_2$(sets)
  - tag_bits = word_size - index_bits - offset_bits
- Way to get offset/index/tag from address (bitwise & numeric):
  - block_offset = addr & ones(offset_bits) = addr % block_size
  - index = (addr >> offset_bits) & ones(index_bits)
    = (addr / block_size) % sets
  - tag = addr >> (offset_bits+index_bits) = addr / (sets*block_size)

ones(n) = a string of *n* ones = ((1<<n)-1)

Cache Replacement Policies

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- Some options
  - **Random**
  - **LRU (least recently used)**

    This is what you usually want

    - Fits with temporal locality, LRU = least likely to be used in future
  - **NMRU (not most recently used)**
    - An easier-to-implement approximation of LRU
    - NMRU=LRU for 2-way set-associative caches
  - **FIFO (first-in first-out)**
    - When is this a good idea?

ABCs of Cache Design

- Architects control three primary aspects of cache design
  - And can choose for each cache independently
- A = Associativity
- B = Block size
- C = Capacity of cache

- Secondary aspects of cache design
  - Replacement algorithm
  - Some other more subtle issues we’ll discuss later

Analyzing Cache Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - Easy to identify
  - **Capacity**: miss caused because cache is too small – would've been miss even if cache had been fully associative
    - Consecutive accesses to block separated by accesses to at least N other distinct blocks where N is number of frames in cache
  - **Conflict**: miss caused because cache associativity is too low – would've been hit if cache had been fully associative
    - All other misses

Stores: Write-Through vs. Write-Back

- When to propagate new value to (lower level) memory?
  - **Write-through**: immediately (as soon as store writes to this level)
    - + Conceptually simpler
    - + Uniform latency on misses
    - – Requires additional bandwidth to next level
  - **Write-back**: later, when block is replaced from this level
    - Requires additional "dirty" bit per block → why?
    - + Minimal bandwidth to next level
      - Only write back dirty blocks
    - – Non-uniform miss latency
      - Miss that evicts clean block: just a fill from lower level
      - Miss that evicts dirty block: writeback dirty block and then fill from lower level

Stores: Write-allocate vs. Write-non-allocate

- What to do on a write miss?
  - **Write-allocate**: read block from lower level, write value into it
    - + Decreases read misses
    - – Requires additional bandwidth
    - Use with write-back
  - **Write-non-allocate**: just write to next level
    - – Potentially more read misses
    - + Uses less bandwidth
    - Use with write-through

Example cache trace

Term	Value	Equation
cache size	4096	given
block size	32	given
ways	2	given
frames		cache size / block size
sets		frames / ways
bits:index		$\log_2$(sets)
bits:offset		$\log_2$(block size)
bits:tag		64 minus the above

addr-dec	addr-hex	tag	index	offset	result
38	0026				
30	001E				
62	003E				
5	0005				
2049	0801				
2085	0825				
60	003C				
4130	1022				
2085	0825				

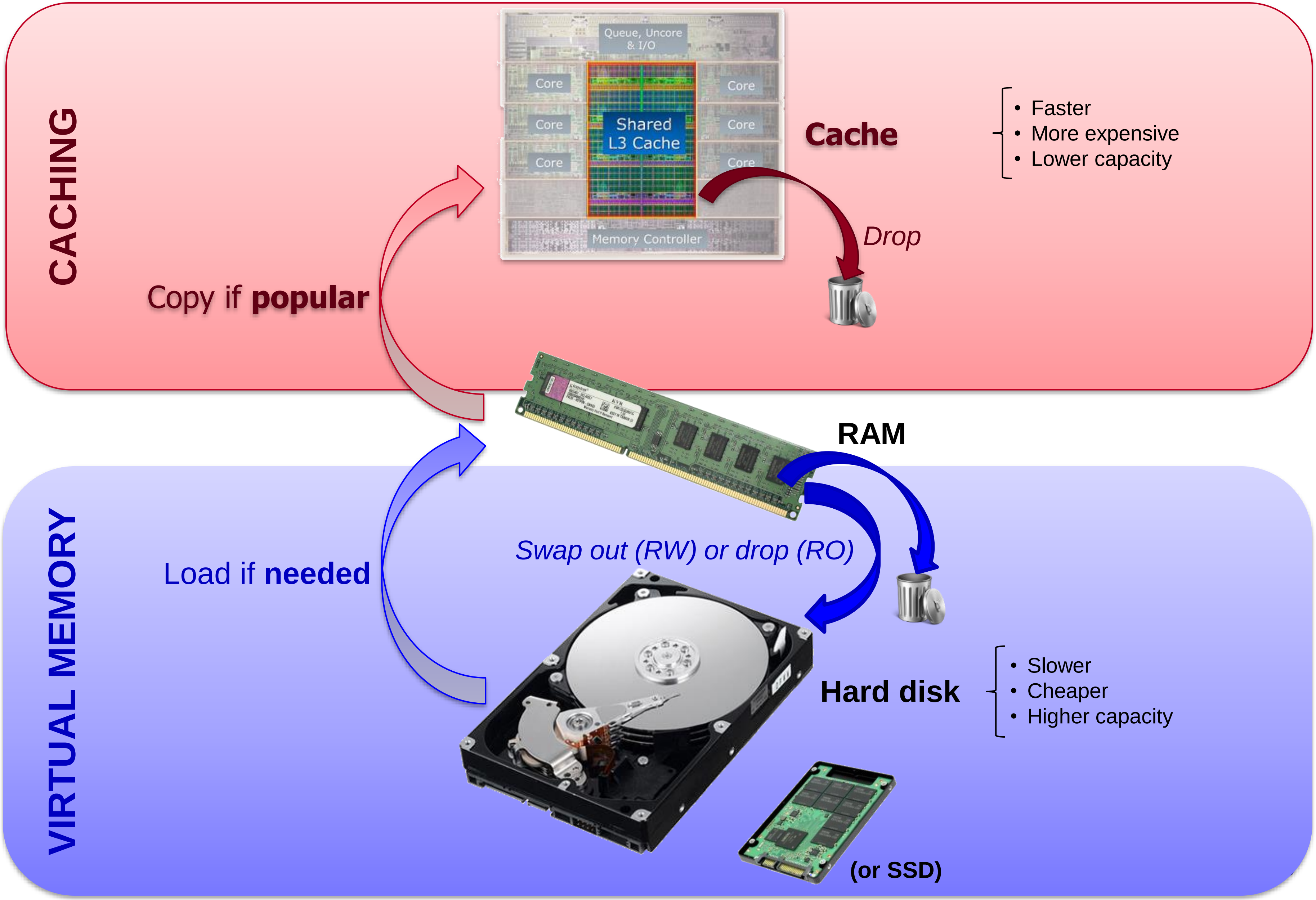
Example cache trace

Term	Value	Equation
cache size	4096	given
block size	32	given
ways	2	given
frames	128	cache size / block size
sets	64	frames / ways
bits:index	6	log$_2$(sets)
bits:offset	5	log$_2$(block size)
bits:tag	53	64 minus the above

addr-dec	addr-hex	tag	index	offset	result
38	0026	0	1	6	miss compulsory
30	001E	0	0	30	miss compulsory
62	003E	0	1	30	hit
5	0005	0	0	5	hit
2049	0801	1	0	1	miss compulsory
2085	0825	1	1	5	miss compulsory
60	003C	0	1	28	hit
4130	1022	2	1	2	miss compulsory
2085	0825	1	1	5	miss conflict

Virtual memory

Figure: caching vs. virtual memory



CACHING

Cache

- Faster
- More expensive
- Lower capacity

Copy if **popular**

Drop

RAM

VIRTUAL MEMORY

Load if **needed**

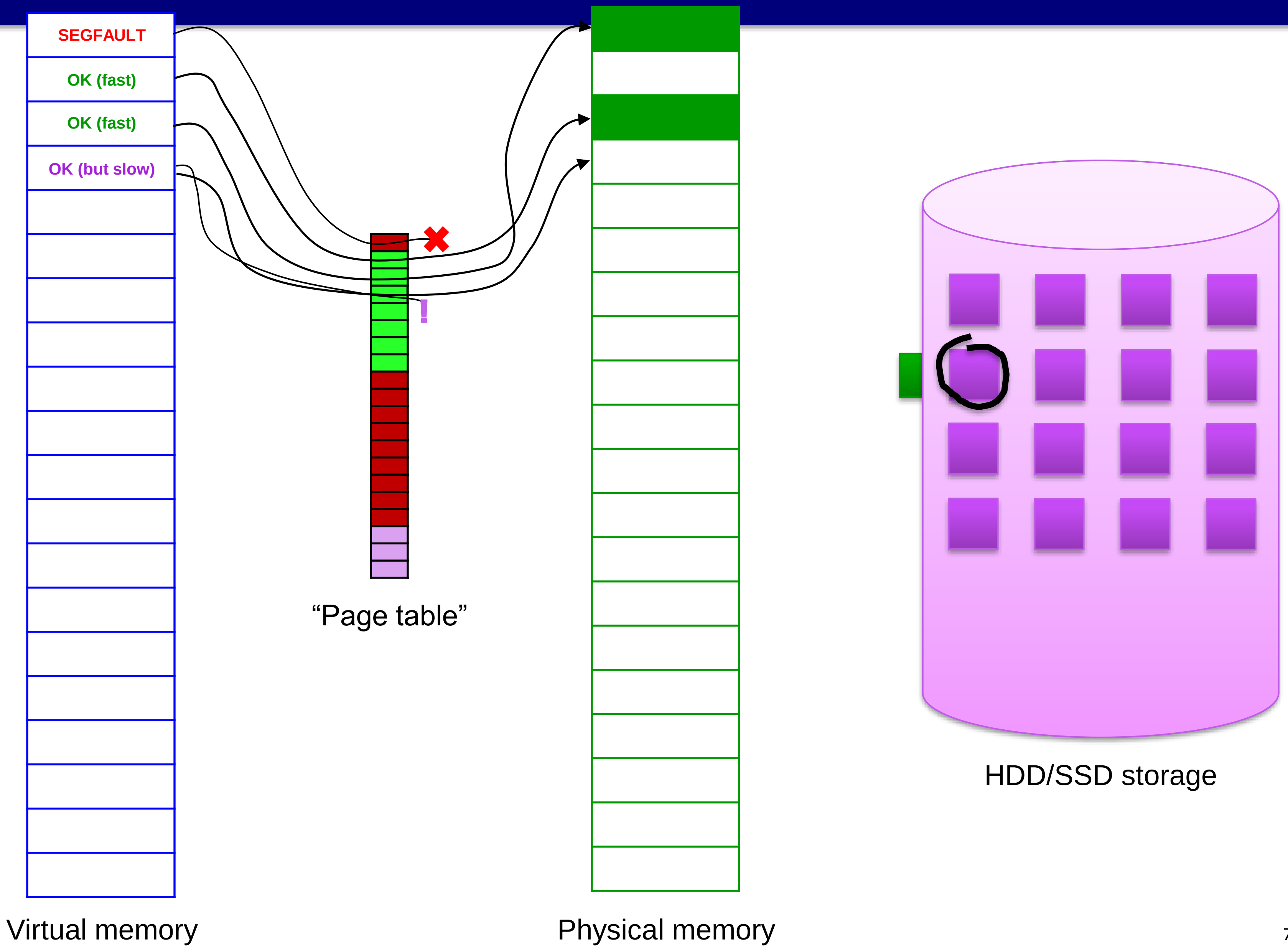
Swap out (RW) or drop (RO)

Hard disk

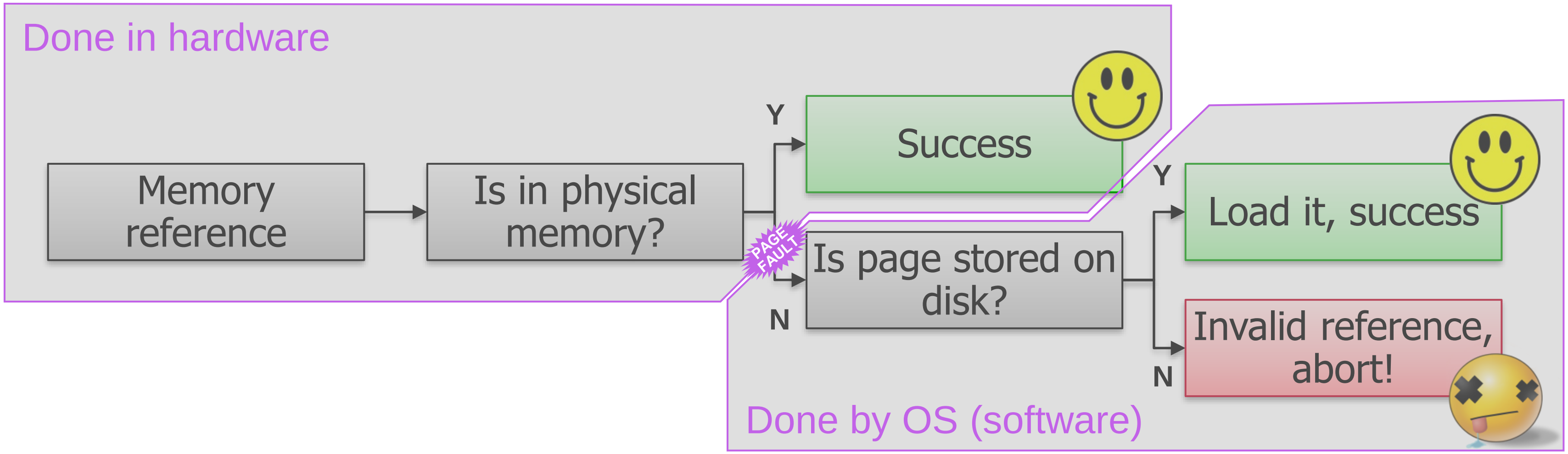
- Slower
- Cheaper
- Higher capacity

(or SSD)

High level operation

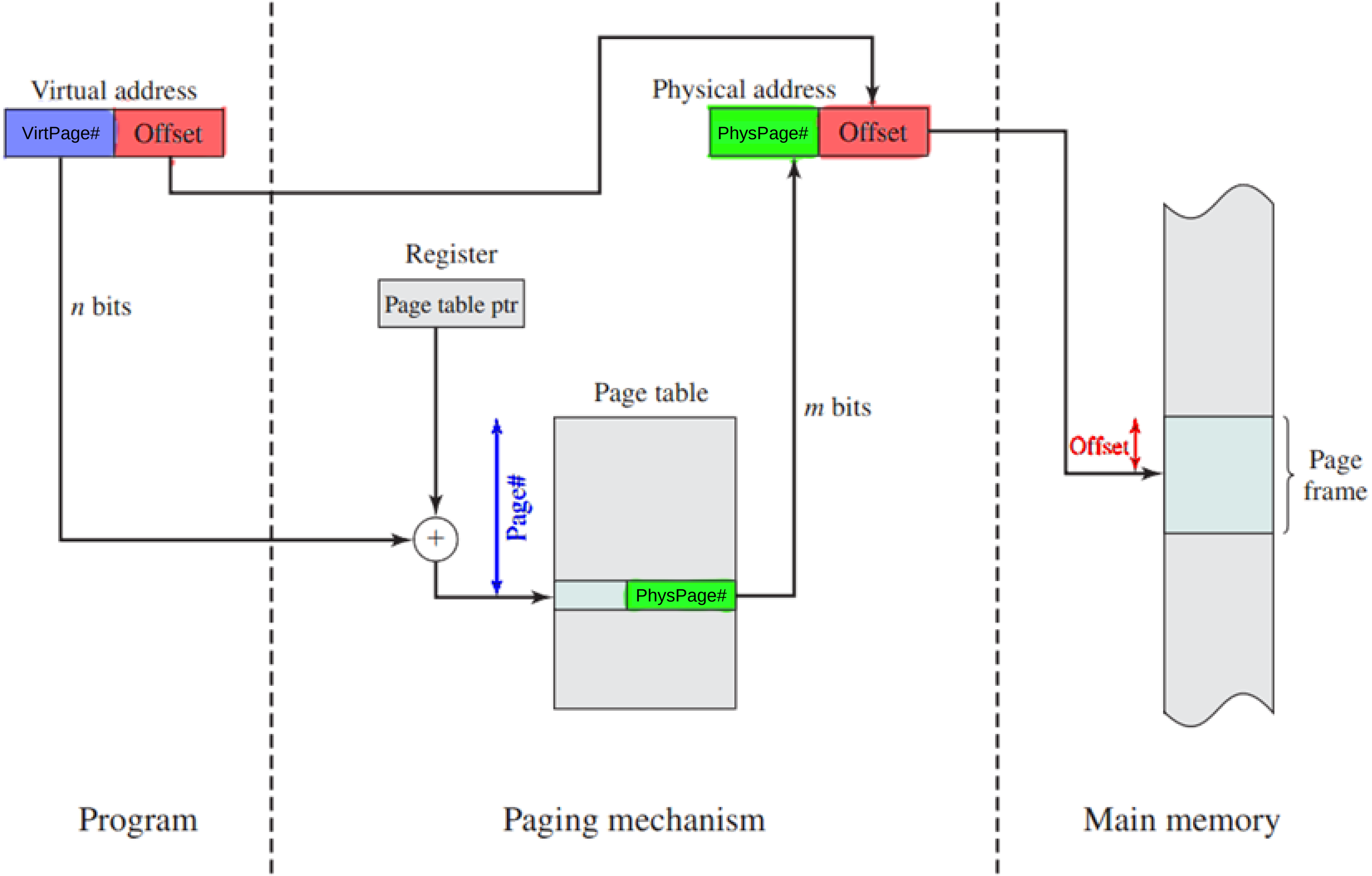


Demand Paging



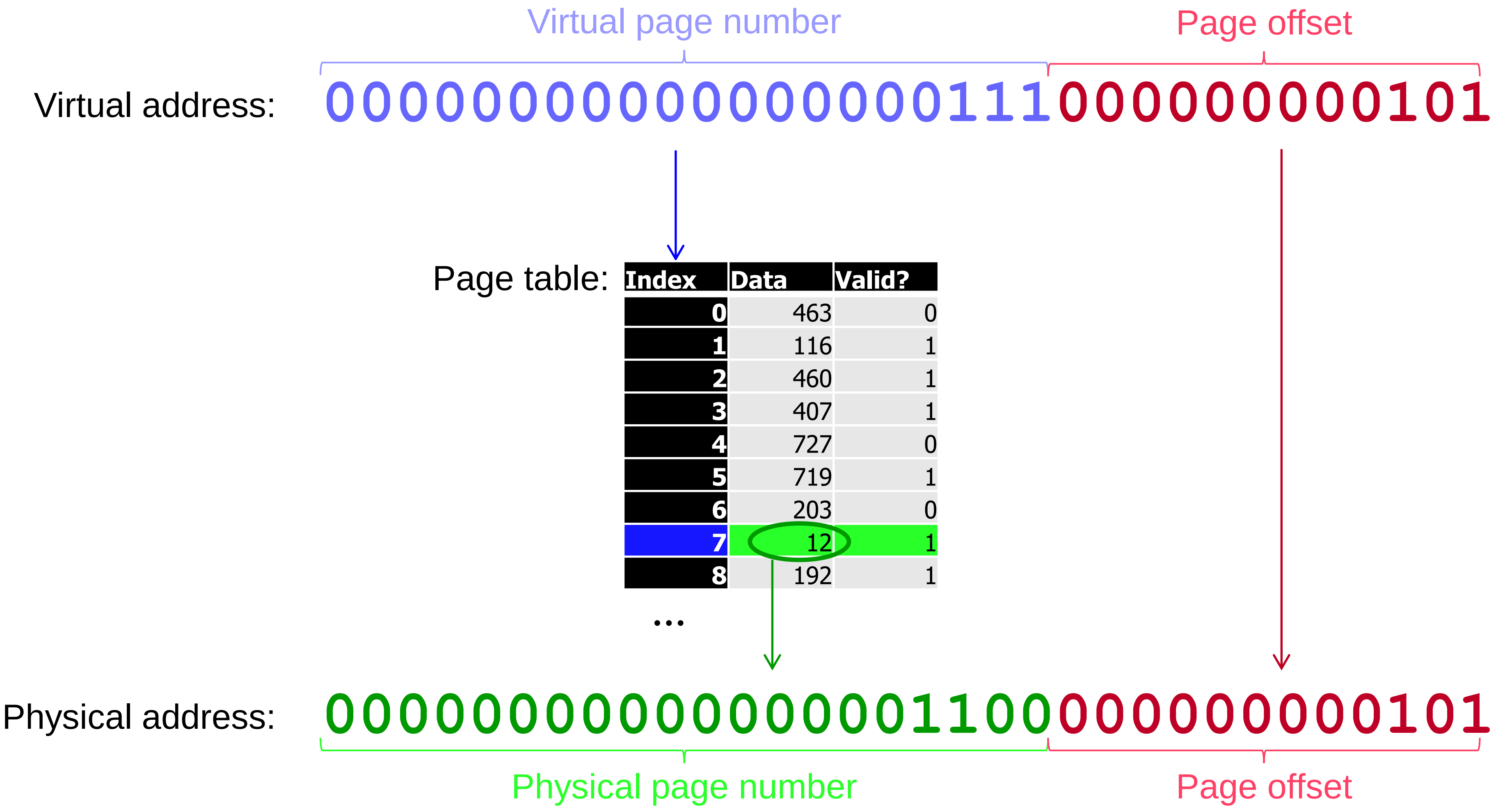
Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

Address translation



Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

Address translation



Virtual page number

Page offset

Virtual address: 00000000000000000111000000000101

Page table:

Index	Data	Valid?
0	463	0
1	116	1
2	460	1
3	407	1
4	727	0
5	719	1
6	203	0
7	12	1
8	192	1

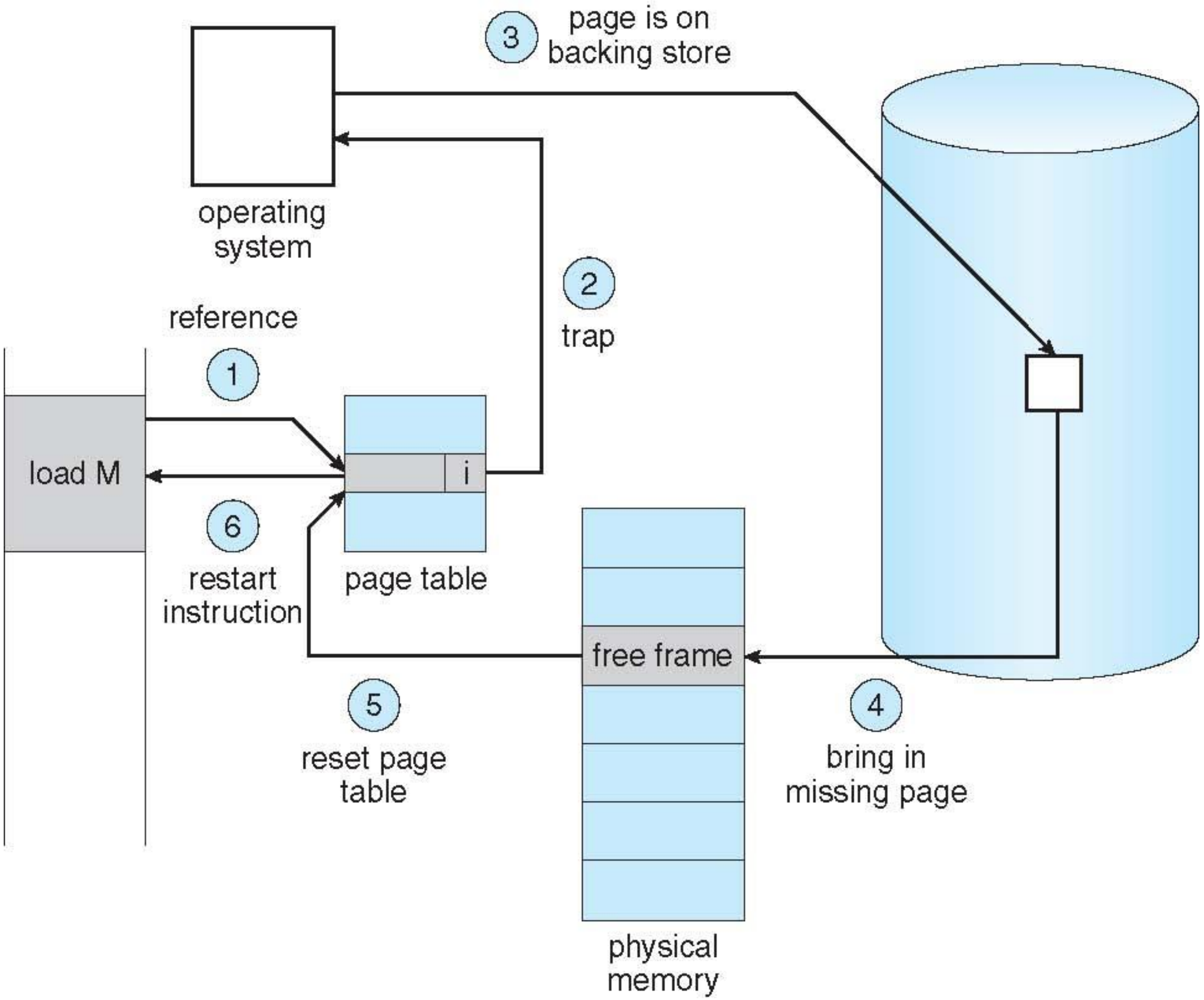
…

Physical address: 00000000000000001100000000000101

Physical page number

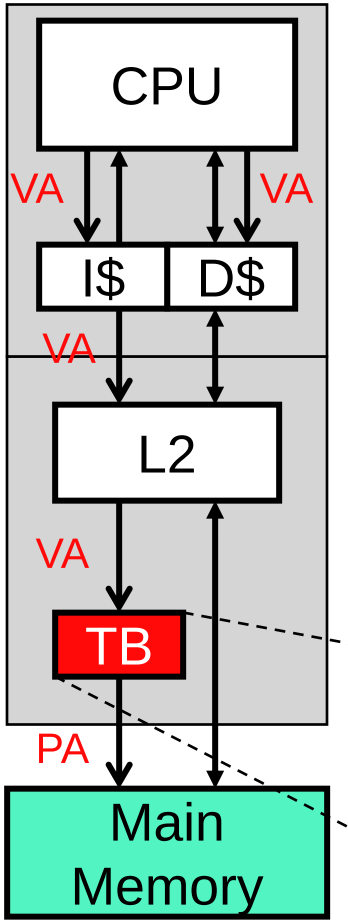
Page offset

Steps in Handling a Page Fault



Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

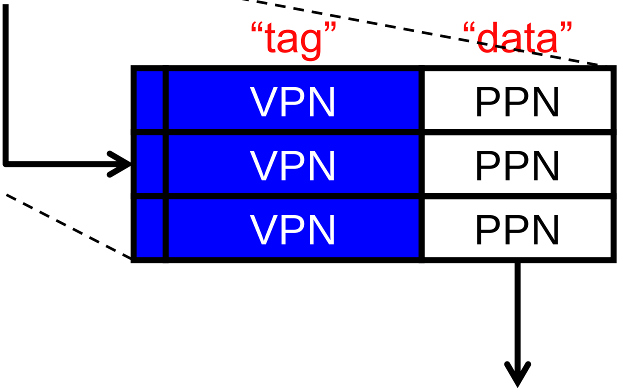
Translation Buffer



- Functionality problem? Add indirection!
- Performance problem? Add cache!

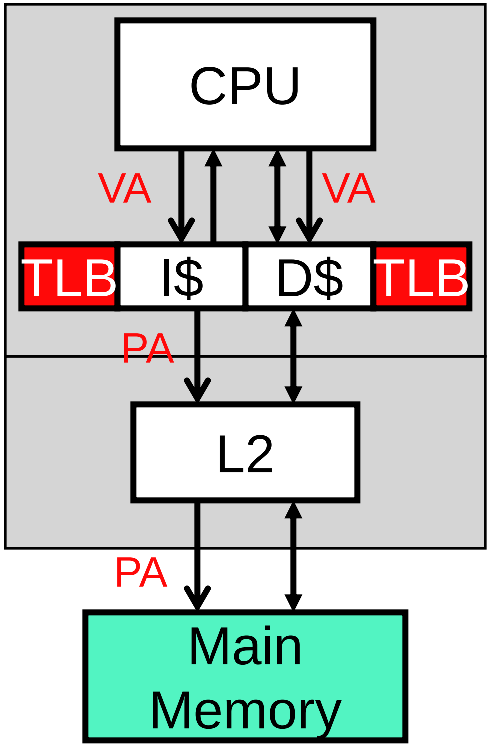
- Address translation too slow?
  - Cache translations in **translation buffer (TB)**
    - Small cache: 16–64 entries, often fully assoc

  + Exploits temporal locality in PT accesses
  + OS handler only on TB miss



© Daniel J. Sorin from Roth

Virtual Physical Caches



Compromise: **virtual-physical caches**

- Indexed by VAs
- Tagged by PAs
- Cache access and address translation in parallel

+ No context-switching/aliasing problems

+ Fast: no additional t_{hit} cycles

- A TB that acts in parallel with a cache is a **TLB**
    - **Translation Lookaside Buffer**

- Common organization in processors today

What Happens if There is no Free Frame?

- **Page replacement** – find *some page* in memory, but not really in use, page it out
  - Algorithm?
  - Want an algorithm which will result in minimum number of page faults
  - *This decision is just like choosing the caching replacement algorithm!*

Same as caching!

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

Thrashing

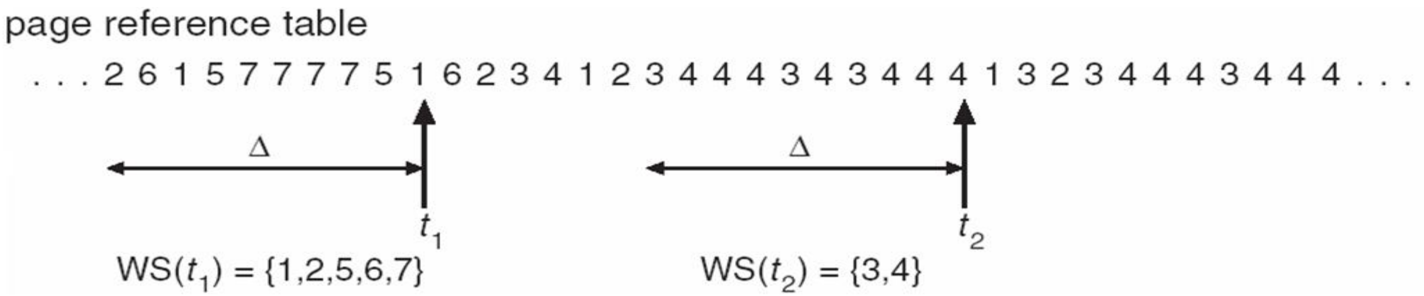
- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- Thrashing ≡ a process is busy swapping pages in and out

Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

Working-set model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes

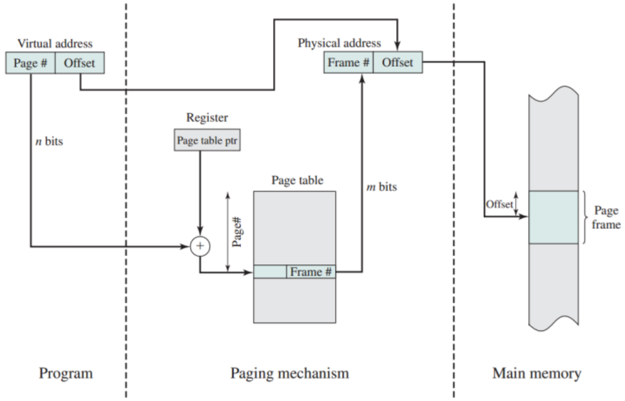


Adapted from Operating System Concepts by Silberschatz, Galvin, and Gagne

Virtual memory summary

WOW!

- Address translation via **page table**
  - Page table turns VPN to PPN (noting the valid bit)
- Page is marked 'i'? **Page fault.**
  - If OS has stored page on disk, load and resume
  - If not, this is invalid access, kill app (seg fault)



- Governing policies:
  - Keep a certain **number of frames loaded** per app
  - Kick out frames based on a **replacement algorithm** (like LRU, etc.)
- Looking up page table in memory too slow, so cache it:
  - The **Translation Buffer (TB)** is a hardware cache for the page table
  - When applied at the same time as caching (as is common), it's called a **Translation Lookaside Buffer (TLB).**
- **Working set size** tells you how many pages you need over a time window.
- **DRAM** is slower than SRAM, but denser. Needs constant refreshing of data.

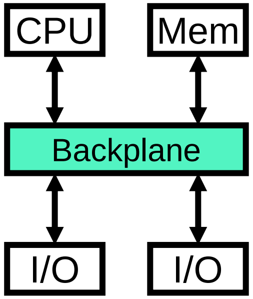
I/O

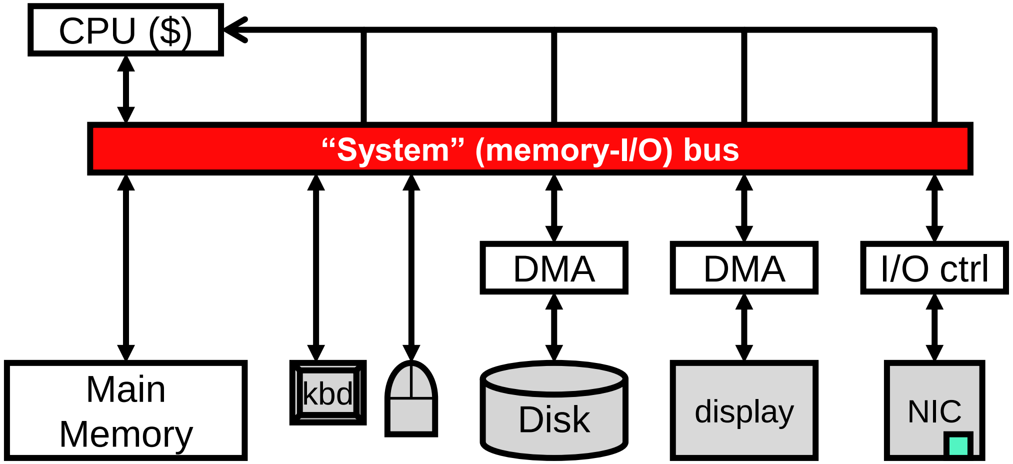
Protection and access

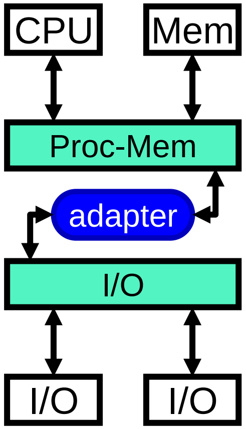
- I/O should be protected, with device access limited to OS
- User processes request I/O through the OS (not directly)
- User processes do so by triggering an **interrupt**, this causes the OS to take over and service the request
- The interrupt/exception facility is implemented in hardware, but triggers OS software

Connectivity

- **Bus**: A communication linkage with two or more devices on it
- Various topologies are possible







Communication models

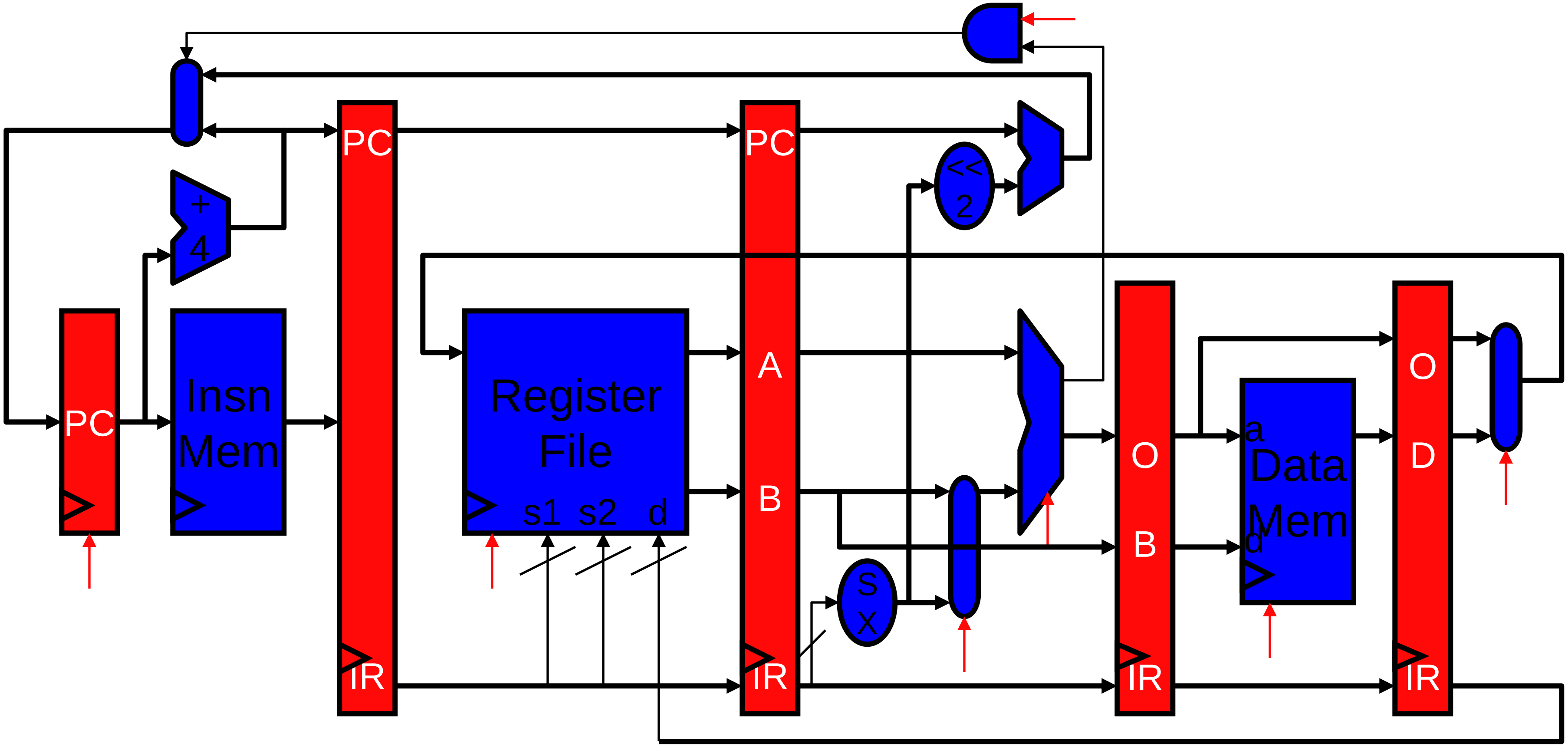
- **Polling**: Ask continuously
  - Often a waste of processor time

- **Interrupts**: Have disk alert the CPU when data is ready
  - But if data packets are small, this interrupt overhead can add up

- **Direct Memory Access (DMA)**: The device itself can put the requested data directly into RAM without the CPU being involved
  - The CPU is alerted via interrupt when the *whole* transaction is done
  - Complication!
    - Now memory can change without notice; interferes with cache
    - Solution: cache listens on bus for DMA traffic, drops changed data

Pipelining

5 Stage Pipelined Datapath



- Temporary values (PC,IR,A,B,O,D) re-latched every stage
  - Why? 5 insns may be in pipeline at once, they share a single PC?
  - Notice, PC not re-latched after ALU stage (why not?)

Pipeline Diagram

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - Convention: **X** means **`lw $4,0($5)`** finishes execute stage and writes into X/M latch at end of cycle 4

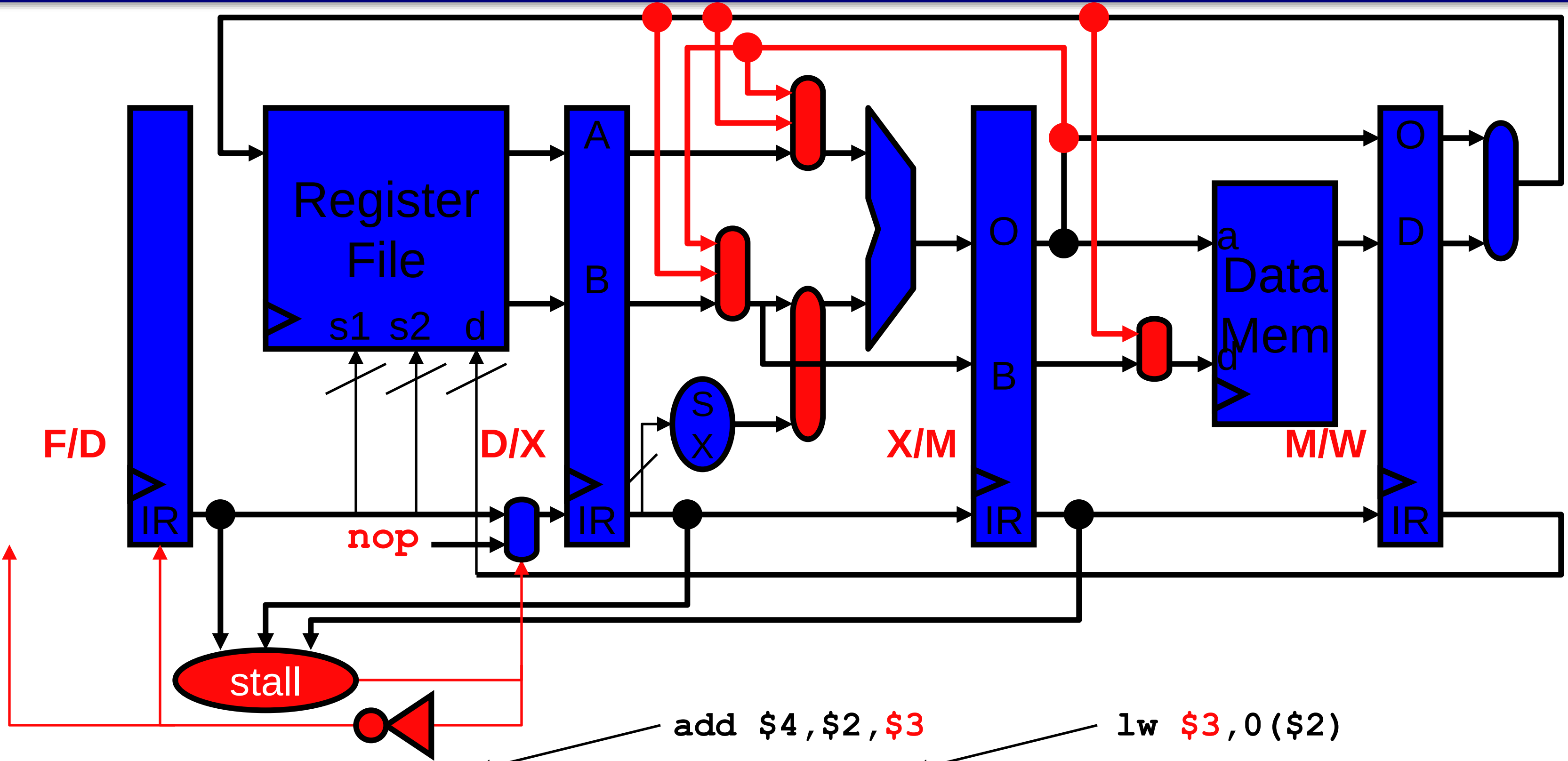
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

Pipeline Hazards

- **Hazard**: condition leads to incorrect execution if not fixed
    - "Fixing" typically increases CPI
    - Three kinds of hazards

- **Structural hazards**
    - Two insns trying to use same circuit at same time
    - Fix by proper ISA/pipeline design:
      Each insn uses every structure exactly once for at most one cycle, always at same stage relative to Fetch
- **Data hazards**
    - Result of dependencies: Need data before it's ready
    - Solve by (a) **stalling** pipeline (inject NOPs) and (b) having **bypasses** provide data before it formally hits destination memory/register.
- **Control hazards**
    - Result of jump/branch not being resolved until late in pipeline
    - Solve by flushing instructions that shouldn't have been happening after branch is resolved
    - This incurs overhead: wasted time! Reduce with:
        - **Fast branches**: Add hardware to resolve branch sooner
        - **Delayed branch**: Always execute instruction after a branch (complicates compiler)
        - **Branch prediction**: Add hardware to speculate on if/where the branch goes

Stalling and Bypassing together



add $4,$2,$3   lw $3,0($2)

add $4,$2,$3   lw $3,0($2)

Stall = (D/X.IR.OP == LOAD) &&

((F/D.IR.RS1 == D/X.IR.RD) ||

((F/D.IR.RS2 == D/X.IR.RD) && (F/D.IR.OP != STORE))

Pipeline Diagram: Data Hazard

- Even with bypasses, stalls are sometimes necessary
- Examples:
  - Memory load -> ALU operation
  - Memory load -> Address component of memory load/store

- Example pipeline diagram for a stall due to a **d**ata hazard:

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	D	X	M	W			
`addi $6,$4,1`			F	**d***	D	X	M	W	

Pipeline Diagram: Control Hazard

- Control hazards indicated with **c*** (or not at all)
  - "Default" penalty for taken branch is 2 cycles:

	1	2	3	4	5	6	7	8	9
`addi $3,$0,1`	F	D	X	M	W				
`bnez $3,targ`		F	D	X	M	W			
`sw $6,4($7)`			**c***	**c***	F	D	X	M	W

  - Fast branches reduce the penalty to 1 cycle:

	1	2	3	4	5	6	7	8	9
`addi $3,$0,1`	F	D	X	M	W				
`bnez $3,targ`		F	D	X	M	W			
`sw $6,4($7)`			**c***	F	D	X	M	W	

Multicore

Types of parallelism

- Pipelining tries to exploit **instruction-level parallelism (ILP)**
  - “How can we simultaneously do steps in this otherwise sequential process?”
- Multicore tries to exploit **thread-level parallelism**
  - “How can we simultaneously do multiple processes?”

- **Thread**: A program has one (or more) threads of control
  - A thread has its own PC
  - Threads in a program share resources, especially memory

    (e.g. sharing a page table)

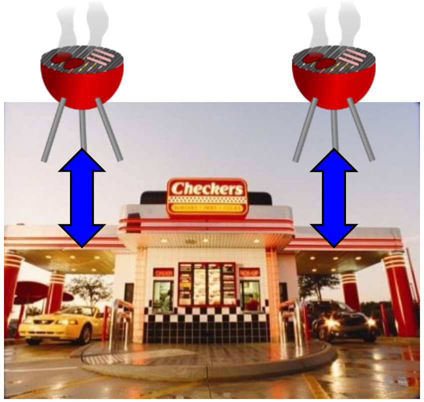
Two cases of multiple threads

- **Multiprogramming**: run multiple programs at once

- **Multithreaded programming**: write software to explicitly take advantage of multiple threads (divide problem into parallel tasks)

Multiprocessors

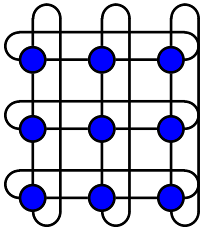
- Multiprocessors: have more than one CPU core
  - Historically: multiple discrete physical chips
  - Now: a single chip with multiple cores

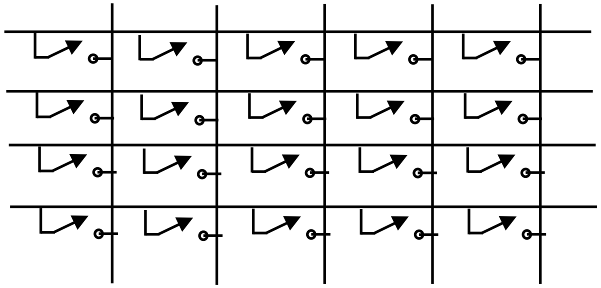


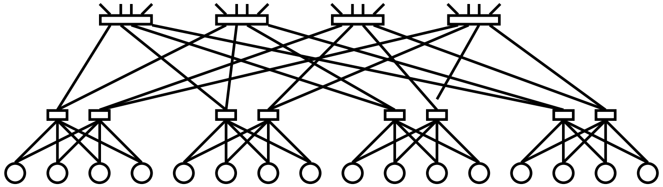
Multiprocessor:
Two drive-throughs, each
with its own kitchen

Challenges of multicore

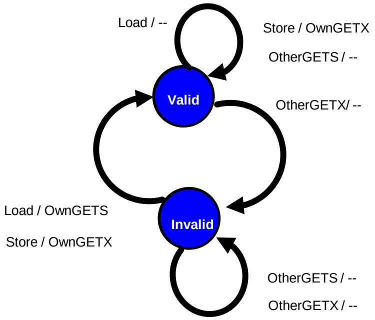
- Two main challenges:
  - **Topologies** of connection (rings, cubes, meshes, buses, etc.)

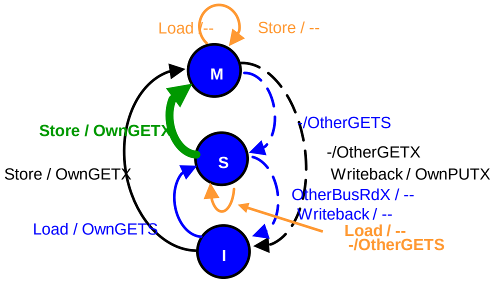






  - **Cache coherence**: If each core has a cache, then each CPU can have a diverging view of memory !! (BAD)
    - Solution: Intelligent caches that use snooping on the memory bus to spot sharing and react accordingly
    - Different coherence algorithms (performance/complexity tradeoffs)





Intel x86

Basic differences

	MIPS	Intel x86
Word size	Originally: 32-bit (MIPS I in 1985) Now: 64-bit (MIPS64 in 1999)	Originally: 16-bit (8086 in 1978) Later: 32-bit (80386 in 1985) Now: 64-bit (Pentium 4's in 2005)
Design	RISC	CISC
ALU ops	Register = Register ⊗ Register (3 operand)	Register ⊗= <Reg|Memory> (2 operand)
Registers	32	8 (32-bit) or 16 (64-bit)
Instruction size	32-bit fixed	Variable: up to 15 *bytes*!
Branching	Condition in register (e.g. "slt")	Condition codes set implicitly
Endian	Either (typically big)	Little
Variants and extensions	Just 32- vs. 64-bit, plus some graphics extensions in the 90s	A bajillion (x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSE4, SSE5, AVX, AES, FMA)
Market share	Small but persistent (embedded)	80% server, similar for consumer (defection to ARM for mobile is recent)

64-bit x86 primer

- Registers:
  - General: `rax rbx rcx rdx rdi rsi r8 r9 .. r15`
  - Stack: `rsp rbp`
  - Instruction pointer: `rip`
- Complex instruction set
  - Instructions are variable-sized & unaligned
- Hardware-supported call stack
  - `call` / `ret`
  - Parameters in registers {`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`}, return value in `rax`
- Little-endian
- These slides use Intel-style assembly language (destination first)
  - GNU tools like `gcc` and `objdump` use AT&T syntax (destination last)

Intel syntax

```
mov   rax, 5
mov   [rbx], 6
add   rax, rdi
push  rax
pop   rsi
call  0x12345678
ret
jmp   0x87654321
jmp   rax
call  rax
```

AT&T syntax

```
mov 5, %rax
mov 6, [%rbx]
add %rdi, %rax
push %rax
pop  %rsi
call 0x12345678
ret
jmp  0x87654321
jmp  %rax
call %rax
```

Binary modification (applies to *all* ISAs)

- Can disassemble binaries (turn into human-readable assembly)
- Do a bunch of cross-referencing to understand functionality (that’s what IDA Pro does)
- Basic blocks of code ending in branches form a flow chart
- Identify behavior and make inferences on author intent

- Can modify by overwriting binary with new instructions (can also *insert* instructions, but this changes layout of binary program, so various pointers have to be updated)

- Cheap and easy technique on x86: overwrite stuff you don’t want with **NOP** (**0x90**)

THE END