Maynooth University
National University of Ireland Maynooth

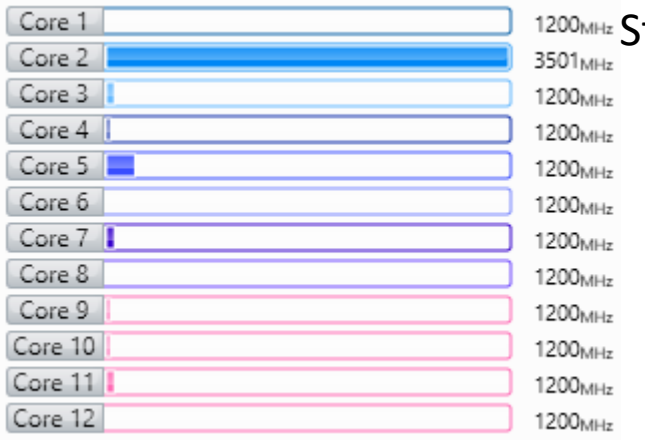
CS433 Modern Architectures

Video 6
Modern Instructions sets – measuring time

Affinity and Forcing an application to run on one core

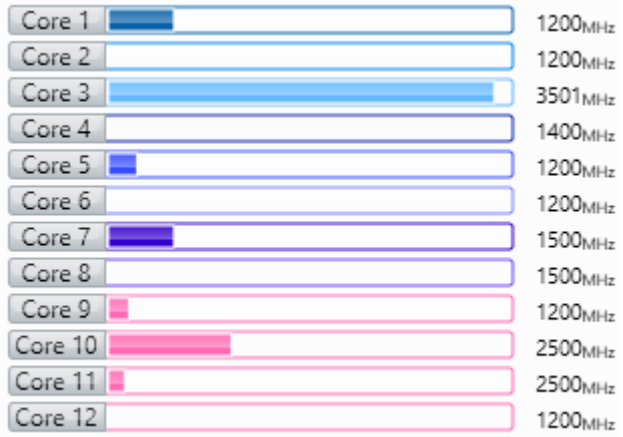
When doing testing on code performance it can be helpful to force it run on one core only. You can do this in windows by creating a text file called, name.bat, and adding the line shown to set the affinity. In this example the same code was run twice. The first run was forced to run on core 2 only. The other changed which core it was running on over the duration of execution.

start /affinity 2 name.exe



Start

start name.exe

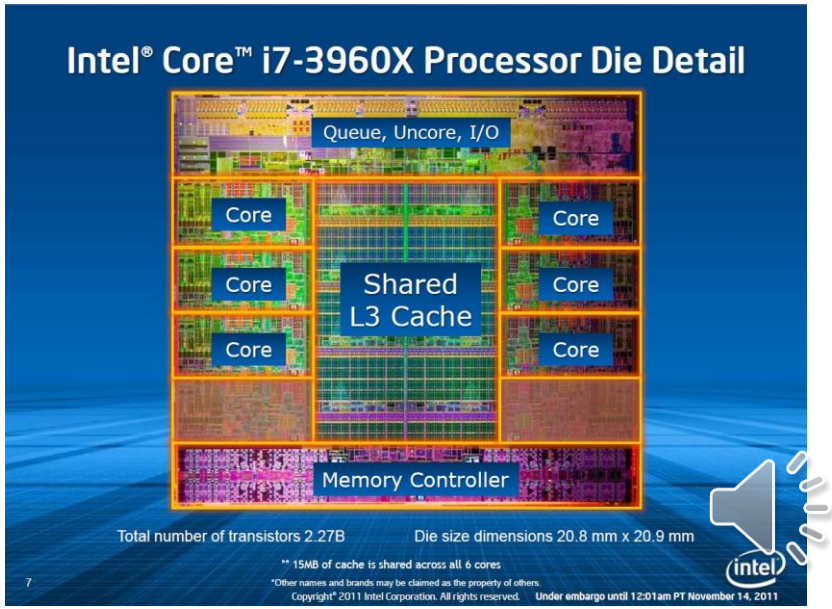


End

Core: A core is a single computing component with an independent CPU.

Thread: A sequence of instructions (code).

Hyperthreading: A core with two sets of registers, that can appear to run two threads simultaneously using features of the pipelining process.



Cache

L1 and L2 are specific to each core, L3 is shared between cores

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

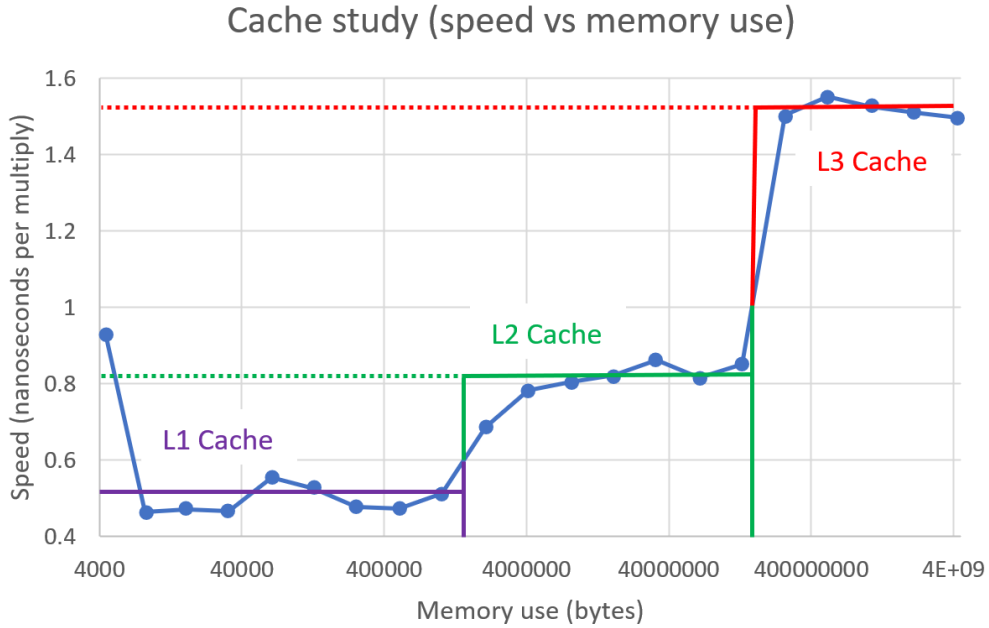
$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + a_{13}z \\ a_2yx + a_{22}y + a_{23}z \\ a_{31}x + a_{32}y + a_{33}z \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = [\vec{a}_1 \quad \vec{a}_2 \quad \vec{a}_3] \begin{bmatrix} x \\ y \\ z \end{bmatrix} = [x\vec{a}_1 + y\vec{a}_2 + z\vec{a}_3]$$

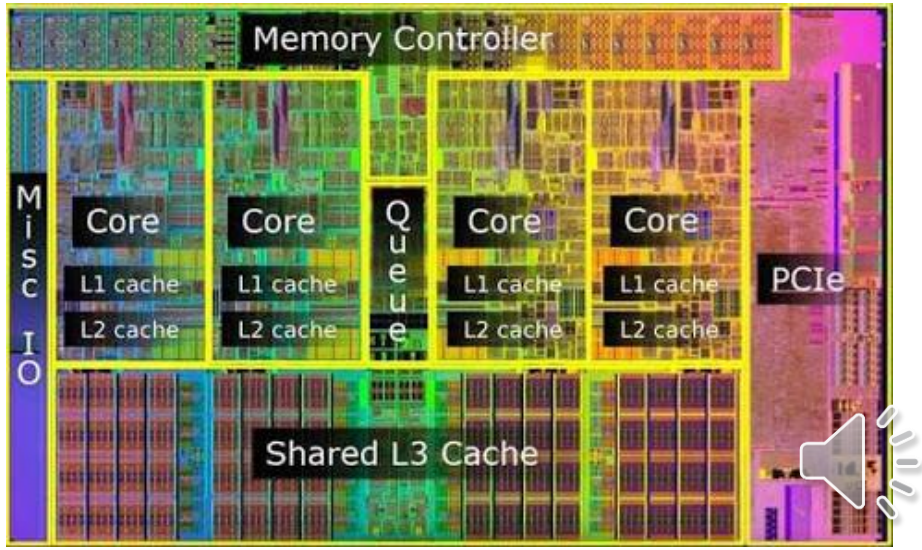
Examples of matrix multiplication

```
QueryPerformanceCounter(&t1);

for (i = 0; i < AR; i++)
for (j = 0; j < BC; j++)
for (k = 0; k < BR; k++)
C[(BC * i) + j] += A[(AC * i) + k] * B[(BC * k) + j];

QueryPerformanceCounter(&t2);

long long memory = ((AR * AC) * sizeof(float)) + ((BR * BC) * sizeof(float)) + ((AR *
BC) * sizeof(float));
elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000000.0 / frequency.QuadPart;
```

C++ code to multiply matrices
C[AR,BC]=A[AR,AC].B[BR,BC]
BR must equal AC
R=Rows
C=Columns



Intel-Core2-i7-5930K
Level 1 6 x 32 KB (0.2MB) 8-way set associative instruction caches
Level 2 6 x 256 KB (1.5MB) 8-way set associative caches
Level 3 15 MB 20-way set associative shared cache



Comparison Static and Dynamic memory

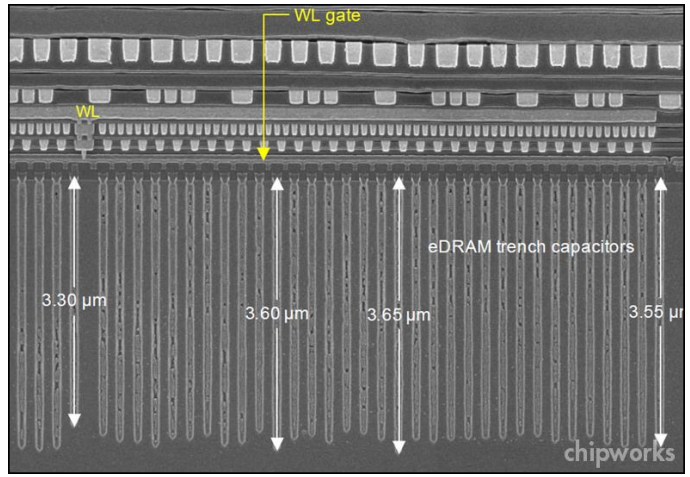
Intel-Core2-i7-5930K

Level 1 6 x 32 KB (0.2MB) 8-way set associative instruction caches – latency 3 clock cycles
Level 2 6 x 256 KB (1.5MB) 8-way set associative caches - latency 10 clock cycles
Level 3 15 MB 20-way set associative shared cache – latency 40 clock cycles

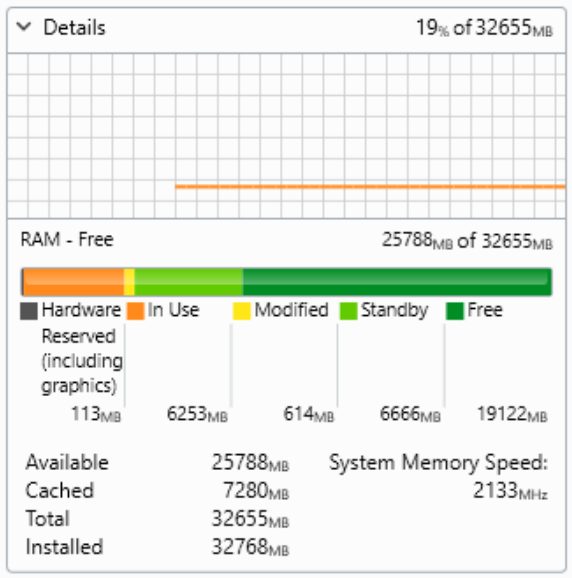
Main memory 32Gbytes DDR4-2666 – 21.3 GB/s – latency – 300 clock cycles

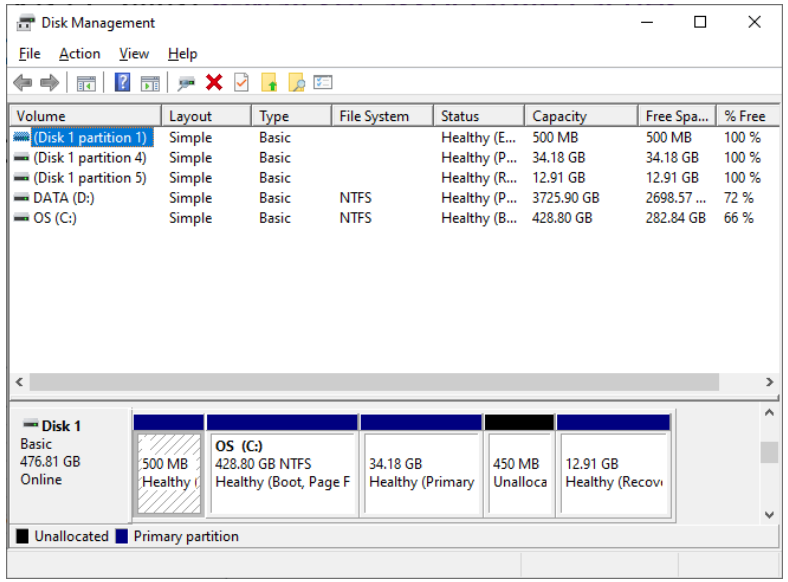
Disks
250Gbyte SSD drive - 560MB/s – latency – 1500 clock cycles
3.6 Tbyte drive - 150MB/s - latency – 10,000,000 clock cycles

DRAM 3.6um x 0.6um

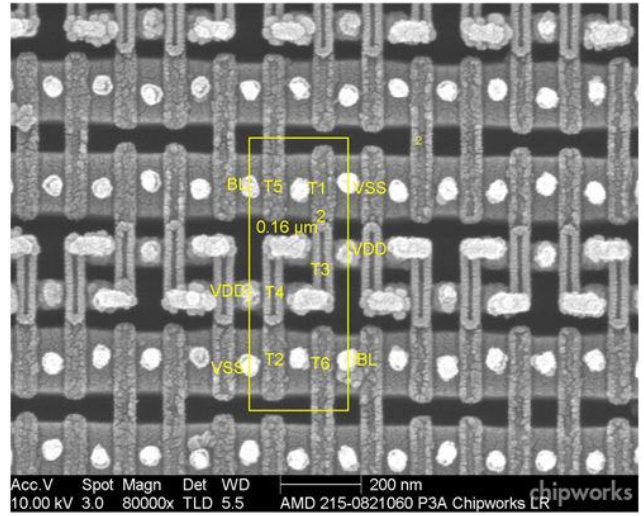


1mm^2=10^12/360*6=46M Apx. 6Mbytes





+ 6 Transistors SRAM 200nm x 500nm



1mm^2=10^12/200*500=10M Apx. 2Mbytes

+ www.chipworks.com

Cache

Locality of reference

Temporal locality (locality of time): If an item is referenced, it will tend to be referenced again soon.

Spatial locality (locality in space): If an item is referenced, nearby items will tend to referenced soon.

Bringing it together

Static ram is fast but low capacity
Dynamic ram is slower but higher capacity

Programs and data are not randomly located

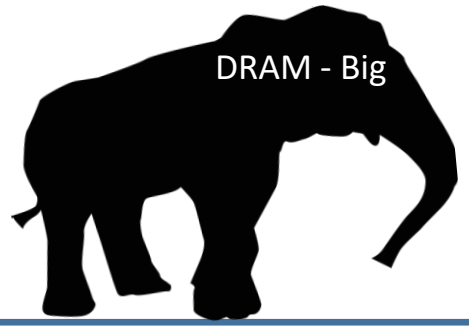
Caching put the frequently used memory in the fast location.

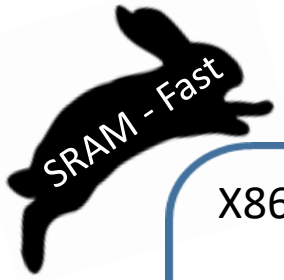
Miss – occurs when the data requested is not in a cache

Compulsory: First read of each block must be from D-RAM, cold start miss.

Capacity: If the number of blocks in regular use exceeds the cache size, then thrashing occurs.

Cache





SRAM - Small

The cache controller contains a cache directory.

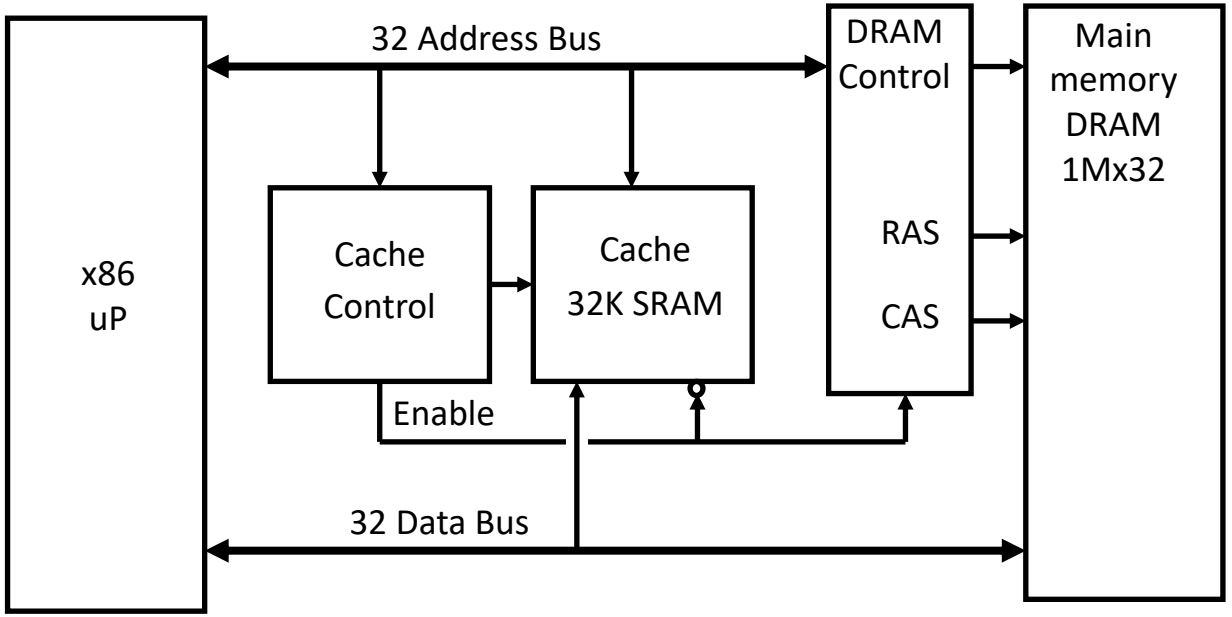
Direct Mapped Cache: The cache is organised to contain a copy of a single page of memory. If processor switches rapidly between pages the effect is known as *thrashing* this will slow the system down.

Two-Way Set Associative: Uses two separate caches to allow fast switching between pages and reduce chance of thrashing.

8-Way Set associate: Uses 8 separate caches.

Fully Associative: Each d-ram address and associated data is stored in cache memory as it is used. Could be slow since you need to look through the directory. How do you discard old data in cache?

X86 Memory system (simplified)



Effective access time (estimate)

10ns static ram, 100nS dynamic ram, 5% miss rate

EAT = (1 – p) x 10 + p (100)

=10(1-P) + 100P

= 9.5 + 5 =14.5nS

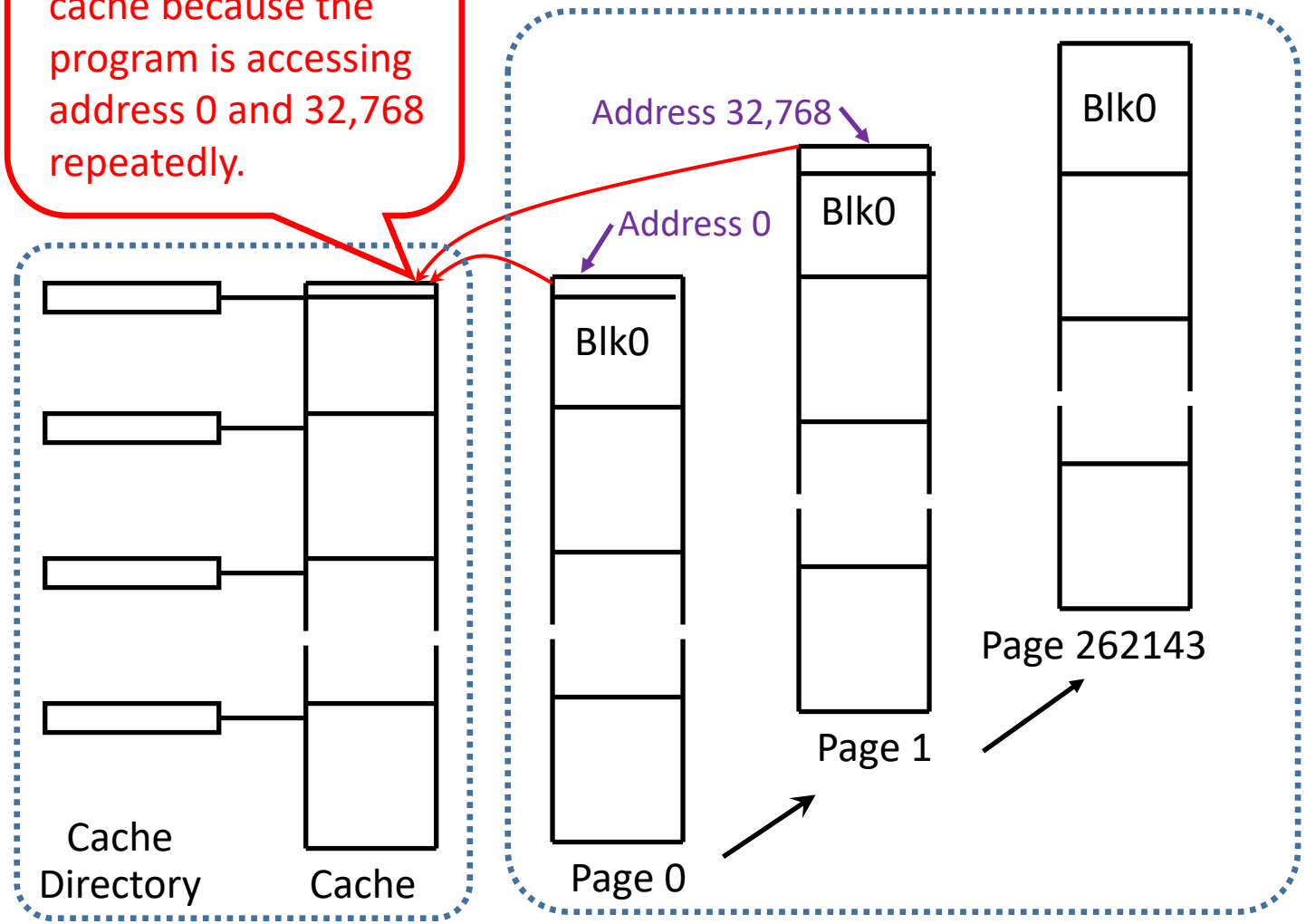
Thrashing

If a program accesses page 0 block X and a then page 1 block X as part of a repeating loop. Each memory access will result in the DRAM access and an update of the cache.

This will delay memory access.

The effect is known as *thrashing*.

Thrashing – keeps swapping data into the cache because the program is accessing address 0 and 32,768 repeatedly.

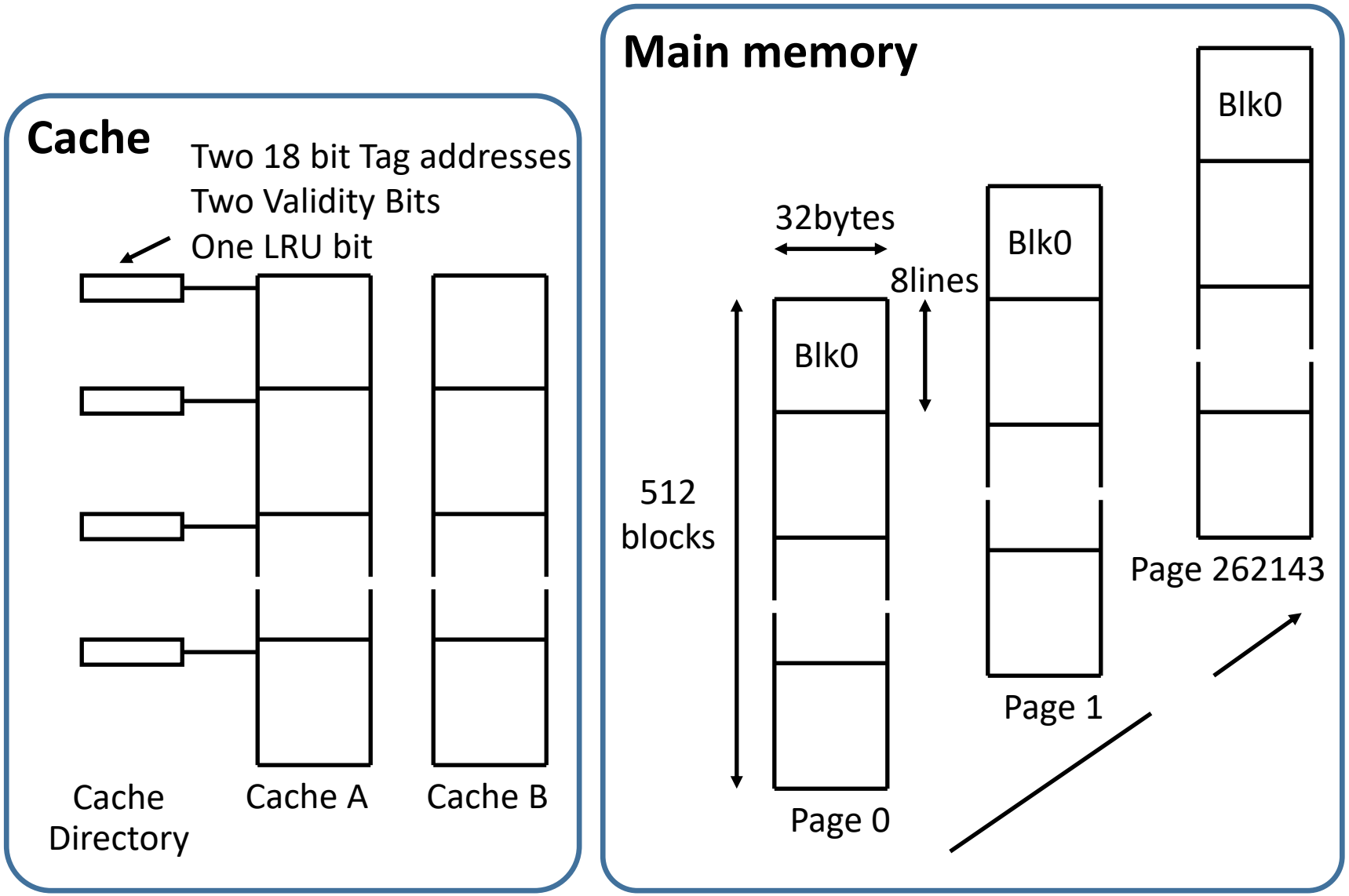


Direct Mapped Cache: for each block of memory there is only one block stored in cache.

Cache directory stores page numbers and validity bit.

Direct mapped caches are prone to thrashing, the solution is a set associative cache which can store two or more blocks in the cache.

Two-Way Set Associative



On a PC the thrashing problem can be solved by splitting the cache into two (or more parts).

Each cache has its own directory.

Additional bit(s) in the tag directory identifie which cache contains the most up to date version in the case both caches contain the same page.

LRU least recently used bit.

Maynooth University
National University of Ireland Maynooth

CS433 Modern Architectures

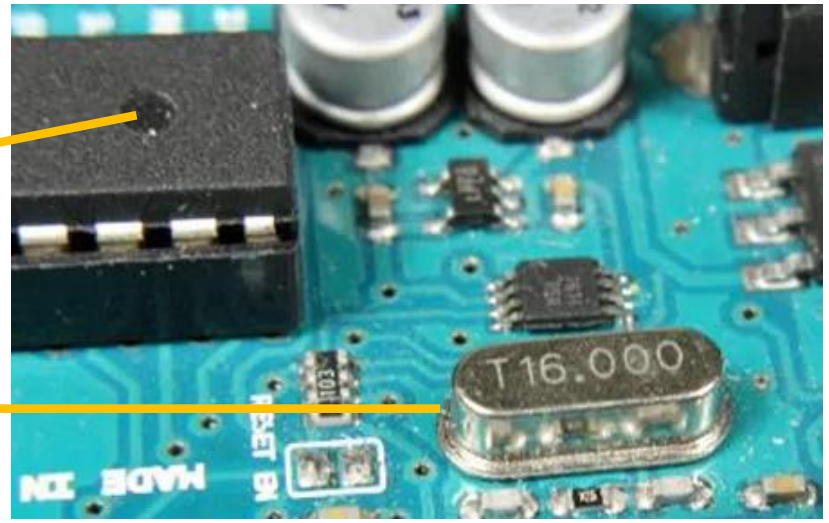
Video 7a

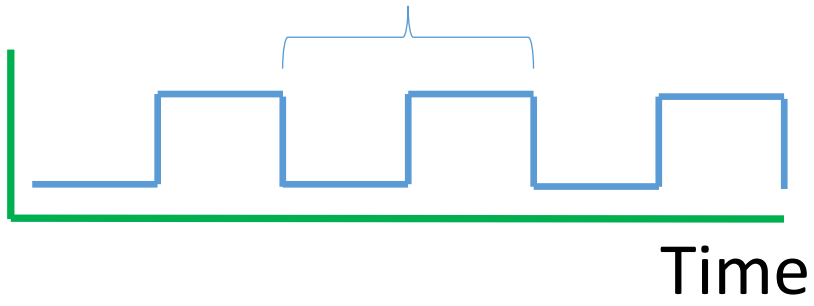
Measuring time

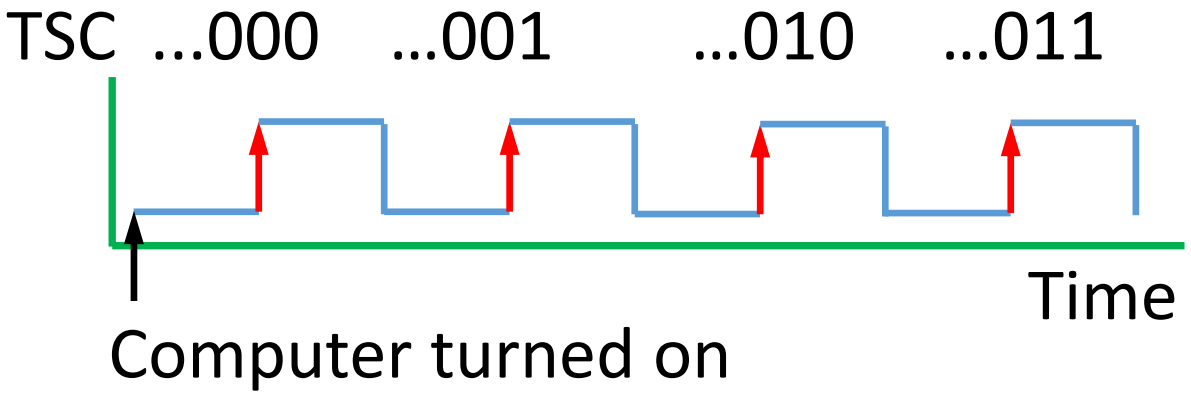
Created for streaming using Panopto within MU Moodle only.

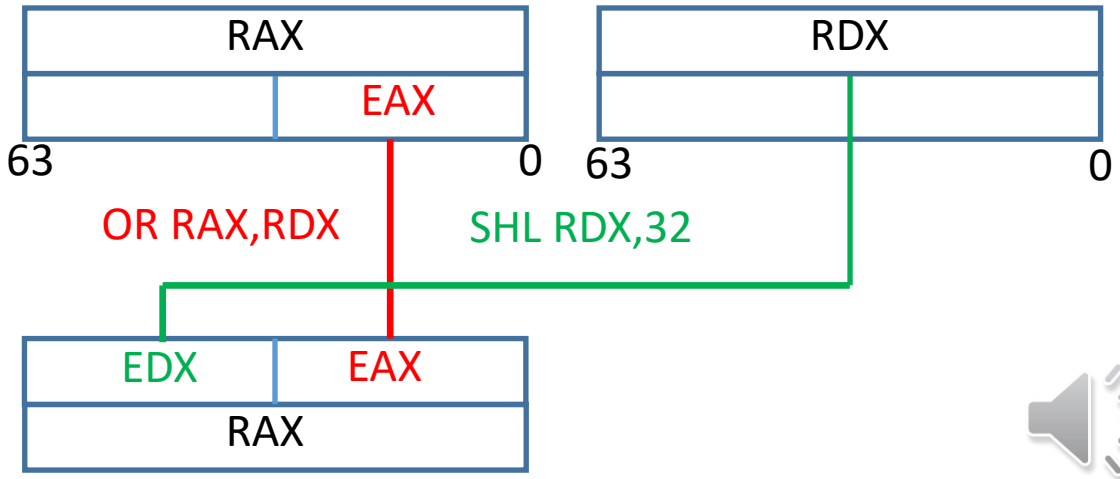
Time Stamp Counter RDTSC, RDTSCP

The time stamp counter is an instruction in the processor that can access a register that is incremented every clock cycle since the CPU started.





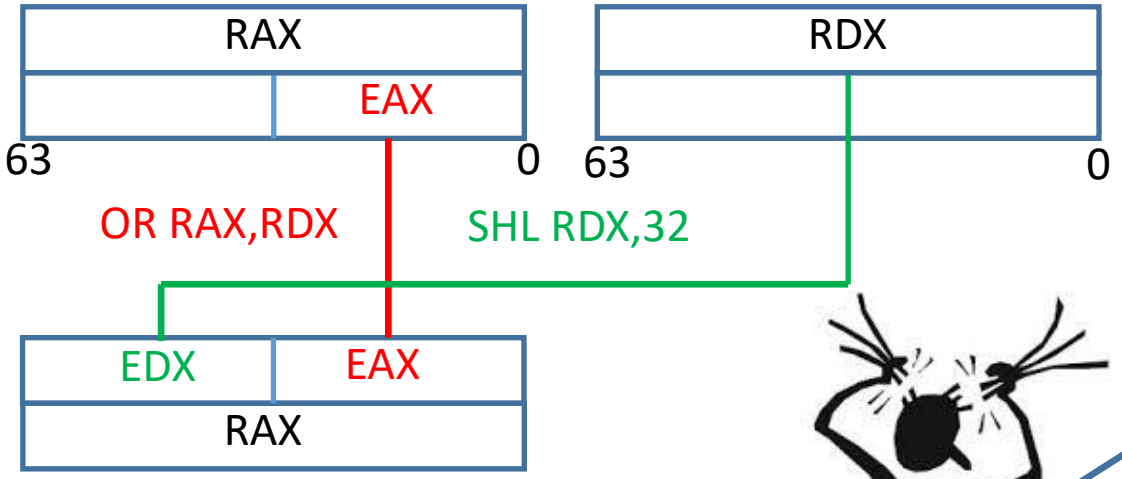




Time Stamp Counter RDTSC, RDTSCP

The traditional description of the machine cycle would make you think the time stamp counter could be a good way to measure time, however it should be used with caution.

The Time Stamp Counter and other diagnostic instructions were introduced when processors started running really fast >66Mhz. It is difficult use oscilloscopes and logic testers above these frequencies so diagnostics went internal to the device.



Warning, modern CPU's

1: Adjust their speed to prevent overheating.

2: Run instructions out of order.

3: The counter can run around the clock. It is a 64 bit counter running at 3.5Ghz -> 167 Years.

Time Stamp Counter RDTSC, RDTSCP

The time stamp counter instructions comes in two forms

RDTSC : Does not force serialisation

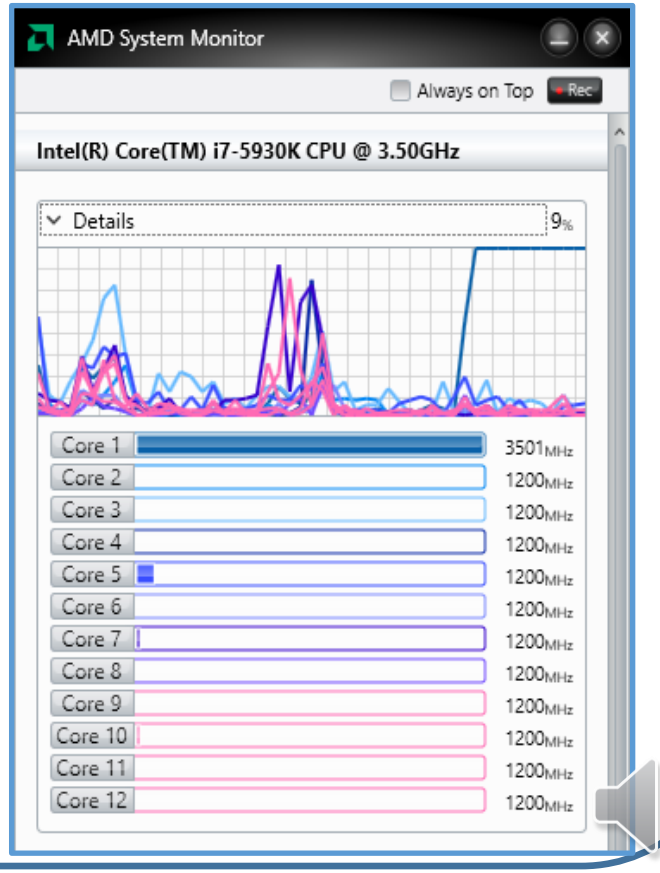
RDTSCP : Forces all instructions to complete prior to its own execution (serialisation)

Instrumentation: adding code to test your program.

When you test code you may wish to limit the cores it runs on using the following line contained in a .bat file.

set /affinity 1 program.exe

Setting the affinity to 7 would run the code across three cores 1+2+4=7.



Time Stamp Counter RDTSC, RDTSCP

C and masm code to access RDTSC

```
#include <stdio.h>

extern "C"
{
   long long readTimeStamp(void);;
}

long long startTime, endTime, elapsedTime;
double duration;

int main()
{
        startTime = readTimeStamp();          // Record start time

        // Code under test

        endTime = readTimeStamp();            // Record stop time
        elapsedTime = endTime - startTime;    // Record cycles used to run code

        duration = ((double)elapsedTime) / (3.5E9)) * 1e9;  // nS, 3.5GHz

        printf("%lld,%lf\n",elapsedTime,duration);     // Estimated time, CPU cycles
for test code

    return 0;
}
```

```
.data
.code

    readTimeStampPROC C

    rdtsc
    shl rdx, 32
    or rax, rdx ; return time in rax
    ret

    readTimeStamp ENDP

end
```

Time Stamp Counter

Estimating the execution time of a single instruction in terms of clock cycles.

```
startTime = readTimeStamp();
for (int i = 0; i < 10000; i++)
{
   total = _mm256_add_ps(total, one);
// total = _mm256_add_ps(total, one);
}
endTime = readTimeStamp();
elapsedTime = endTime - startTime;
duration = (((double)elapsedTime/10000.0)/(3.5E9))*1e9;
printf("%lld clock cycles,%lfns\n",elapsedTime,duration);
```

	_mm256_add_ps(total, one)			_mm256_add_ps(total, one)x2	
0	40704	1.16	0	68104	1.95
1	30188	0.86	1	204676	5.85
2	30180	0.86	2	60032	1.72
3	30032	0.86	3	60028	1.72
4	30104	0.86	4	60028	1.72
5	30260	0.86	5	61400	1.75
6	30176	0.86	6	60028	1.72
7	30032	0.86	7	60028	1.72
8	30100	0.86	8	61004	1.74
9	30036	0.86	9	60636	1.73
Average	30139	0.86		60257	1.73

CPU clocked at 3.5Ghz

(60257-30139)/10000=3.0018 clocks/instruction

Each mm256_add does 8 floating point adds

Processor has 6 cores – running 12 threads

Upper limit 1/ $(3/3.5 \times 10^9)/(8*12)$ = 112GFlops

Maynooth University
National University of Ireland Maynooth

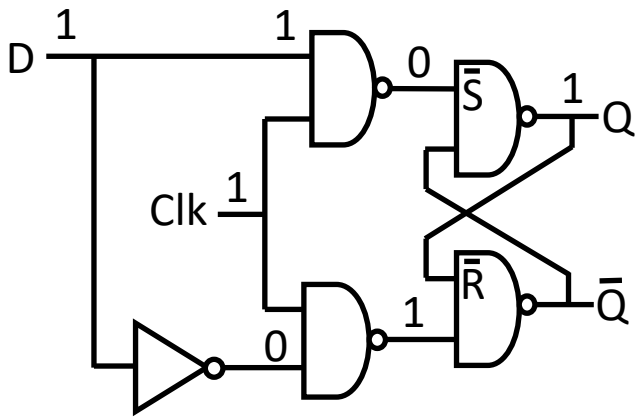
CS433 Modern Architectures

Video 7b

Reminder of clock cycles

Registers (electronic view)

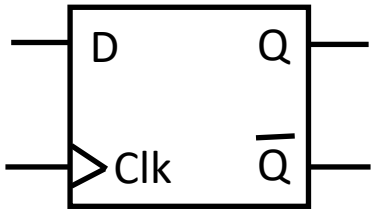
D type flip flop



D-type flip flop (level sensitive)

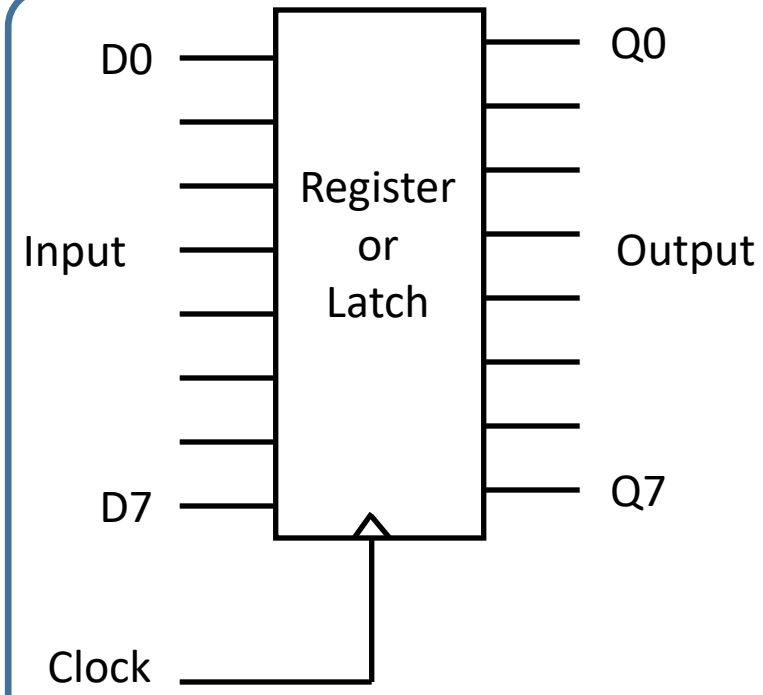
Q=D when Clk=1
Q=last D when Clk=0

Edge triggered D-type flip flop



Clk	D	Q
0	x	No change
1	x	No change
↑	0	0
↑	1	1
↓	x	No change

The D-type flip flop can be used to store a single bit. The input value, D, is transferred to the output, Q on a rising edge of the clock. It remains unchanged until the next rising edge on the clock.



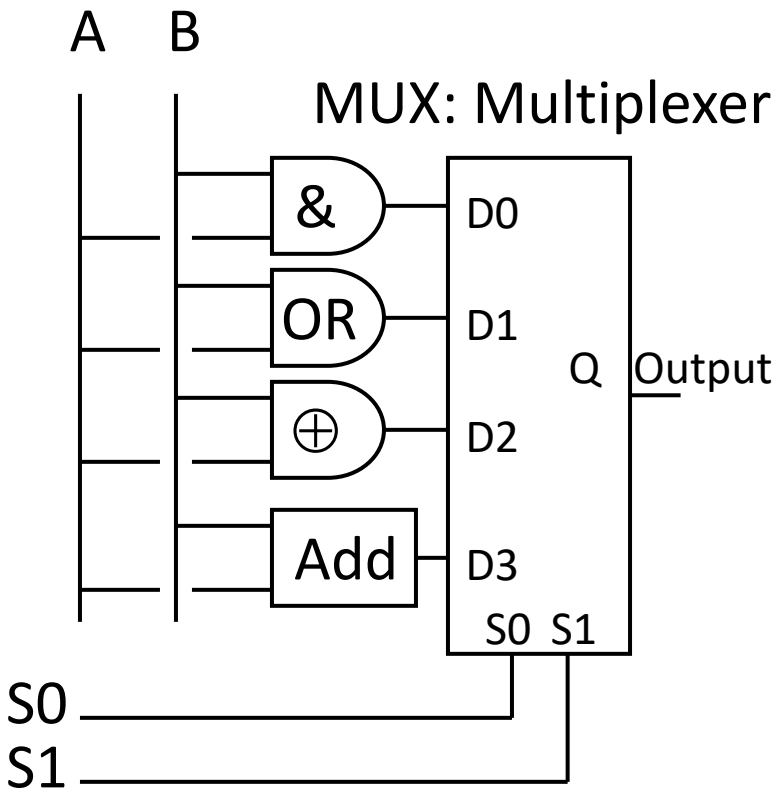
Information (a binary number) is transferred (stored) to the output on next rising edge of the clock.

8 bit latch or register

The CPU contains a number of internal memory locations made of D type latches. These are integral to the CPU. Instructions can address these internal registers directly (register addressing). Each internal memory contains 8, 16, 32, 64 bits (typically) and is known as a register. Each register is designed for a specific purpose, e.g. calculation, stack pointer, instruction pointer.

ALU – Arithmetic logic unit

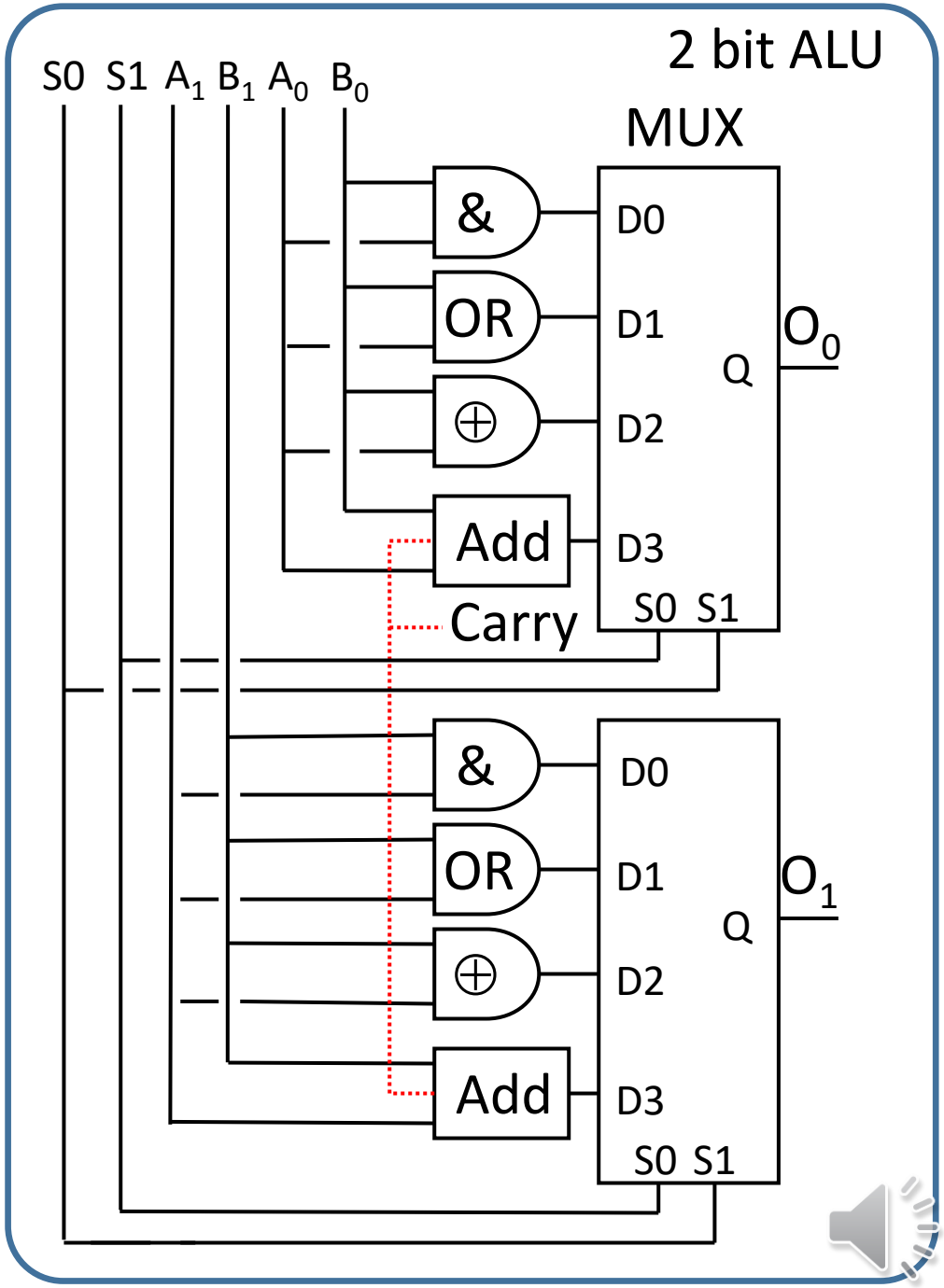
1 bit ALU



S0	S1	Output
0	0	A&B
0	1	AORB
1	0	A⊕B
1	1	A+B

Inputs A and B are operated on by all the functions available. The multiplexer (1 pole 4 way switch) connects the ALU output to the desired function.

2 bit ALU



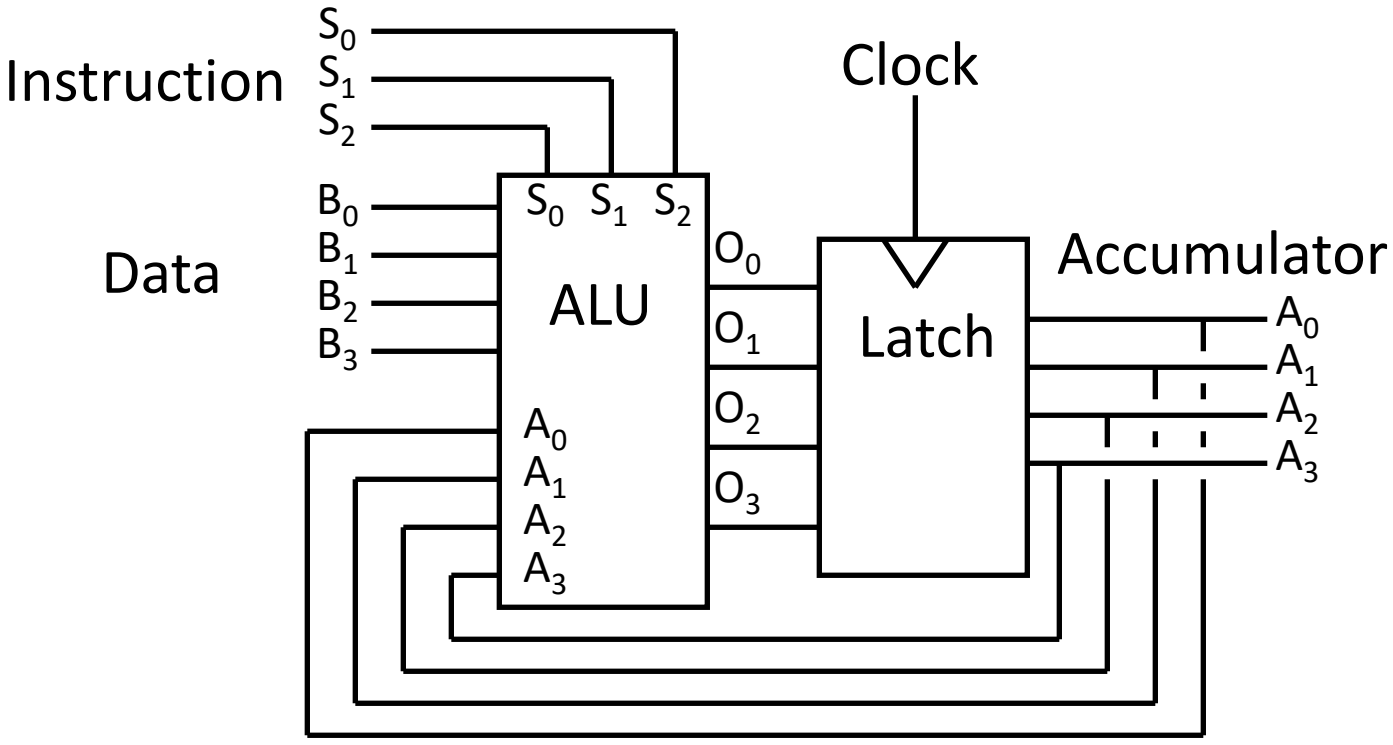
The machine cycle – classical approach

All processors have two key registers

1: Accumulator – this keeps a running total of a calculation

2: Instruction pointer – the line of code (or address) of the instruction being executed.

Calculator combines the ALU and Latch



Operator (mov) puts immediate value in register AX

The x86x64 processor is in its simplest form a *von Neumann* processor, this means it uses the same memory to store the instructions for the program and data for the program.

The commands built into the processor have two parts, the *operator* that specifies the action to be taken and the *operand* which describes the data required to carry out the operation.

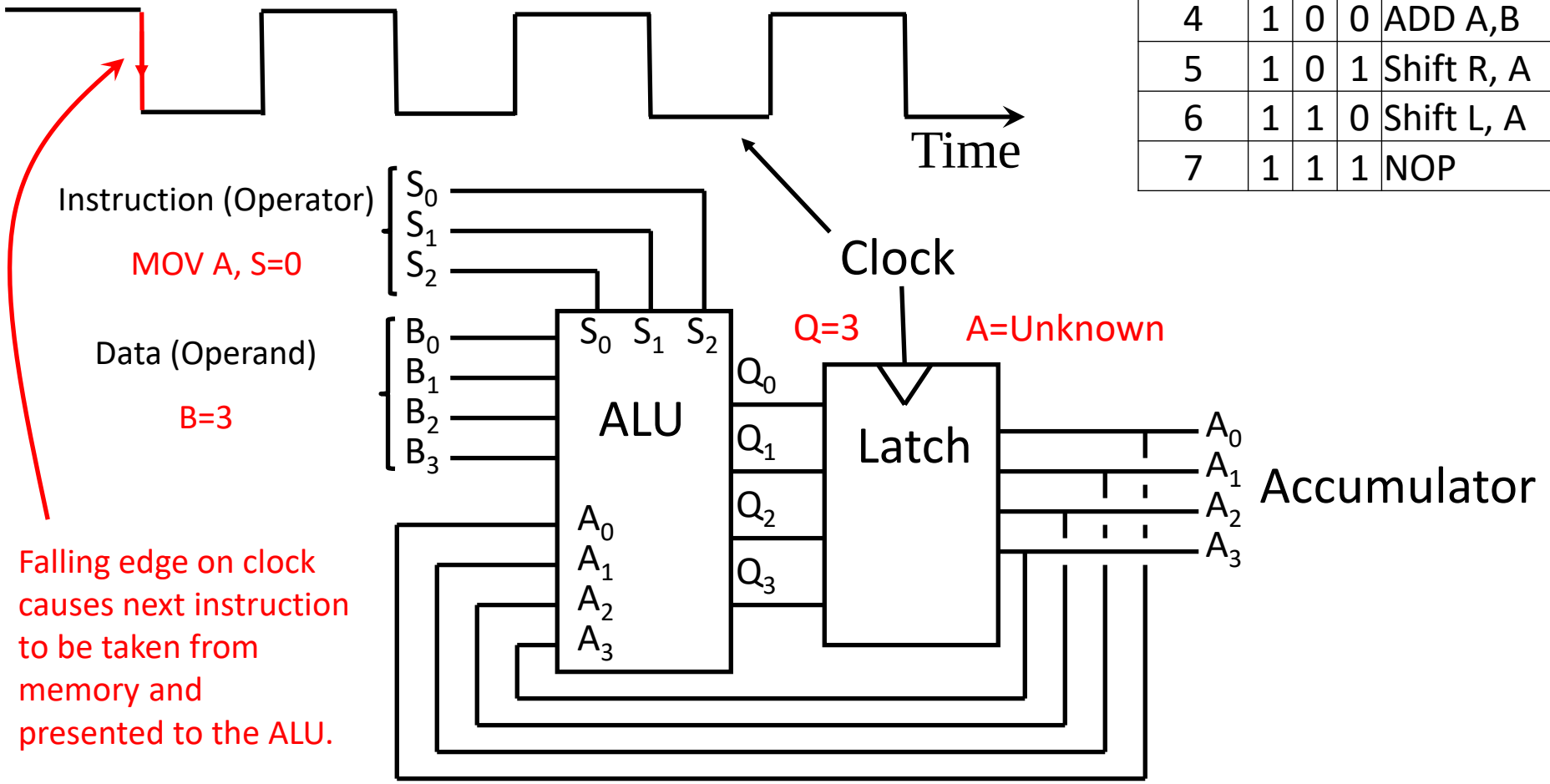
Operator → mov ax,568 ← Operands

Program to evaluate 5-3

Assembly	Machine code	Accumulator
MOV A, 3	S=0, B=3	A=0011b=3
XOR A,15	S=3, B=15	A=1100b=12
ADD A,1	S=4, B=1	A=1101b=13=-3 TC
ADD A,5	S=4, B=5	A=0010b=2

ALU Instruction set

S(Hex)	S2	S1	S0	Function
0	0	0	0	MOV A,N
1	0	0	1	AND A,B
2	0	1	0	OR A,B
3	0	1	1	XOR A,B
4	1	0	0	ADD A,B
5	1	0	1	Shift R, A
6	1	1	0	Shift L, A
7	1	1	1	NOP

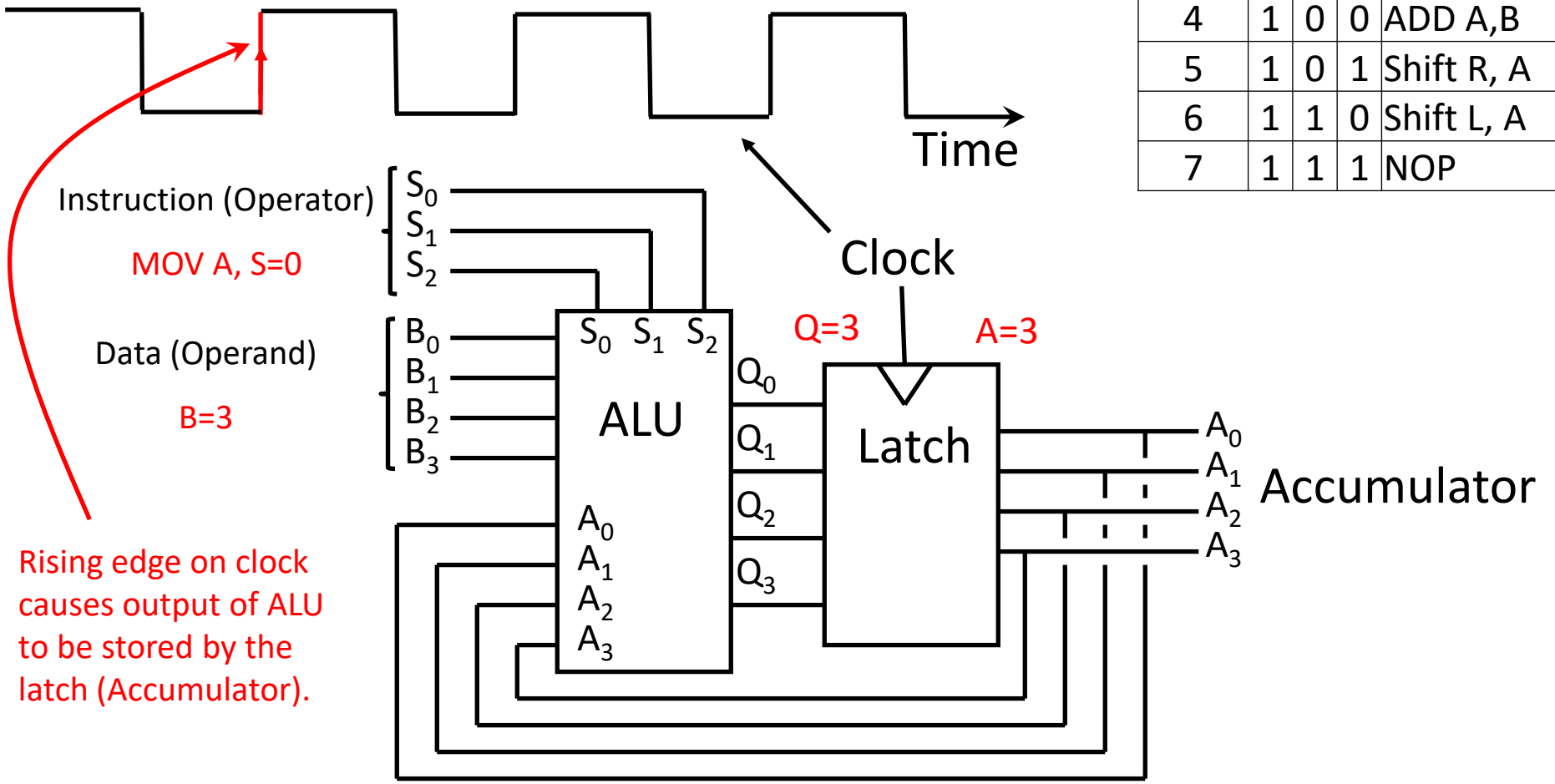


Program to evaluate 5-3

Assembly	Machine code	Accumulator
MOV A, 3	S=0, B=3	A=0011b=3
XOR A,15	S=3, B=15	A=1100b=12
ADD A,1	S=4, B=1	A=1101b=13=-3 TC
ADD A,5	S=4, B=5	A=0010b=2

ALU Instruction set

S(Hex)	S2	S1	S0	Function
0	0	0	0	MOV A,N
1	0	0	1	AND A,B
2	0	1	0	OR A,B
3	0	1	1	XOR A,B
4	1	0	0	ADD A,B
5	1	0	1	Shift R, A
6	1	1	0	Shift L, A
7	1	1	1	NOP

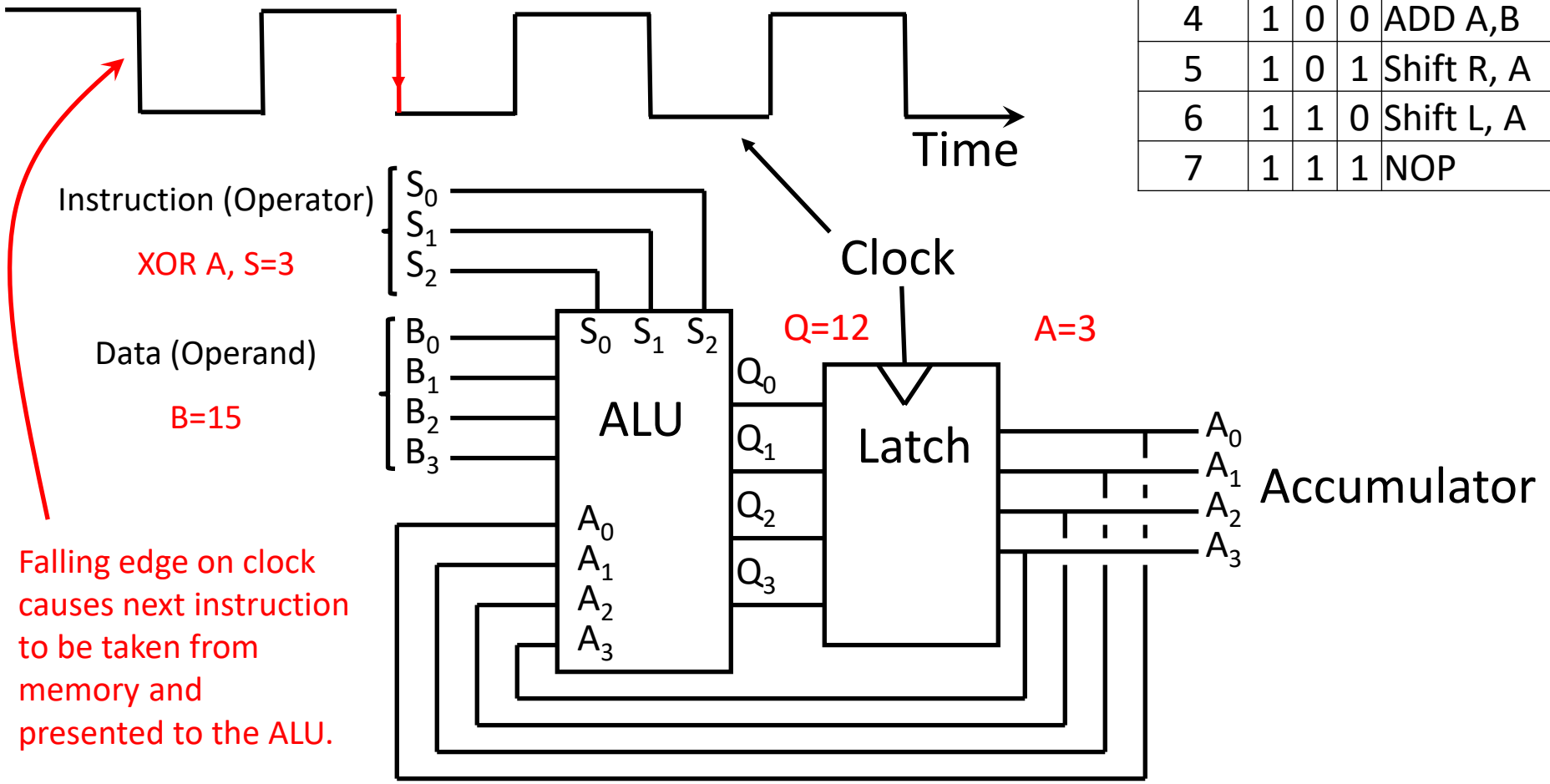


Program to evaluate 5-3

Assembly	Machine code	Accumulator
MOV A, 3	S=0, B=3	A=0011b=3
XOR A,15	S=3, B=15	A=1100b=12
ADD A,1	S=4, B=1	A=1101b=13=-3 TC
ADD A,5	S=4, B=5	A=0010b=2

ALU Instruction set

S(Hex)	S2	S1	S0	Function
0	0	0	0	MOV A,N
1	0	0	1	AND A,B
2	0	1	0	OR A,B
3	0	1	1	XOR A,B
4	1	0	0	ADD A,B
5	1	0	1	Shift R, A
6	1	1	0	Shift L, A
7	1	1	1	NOP

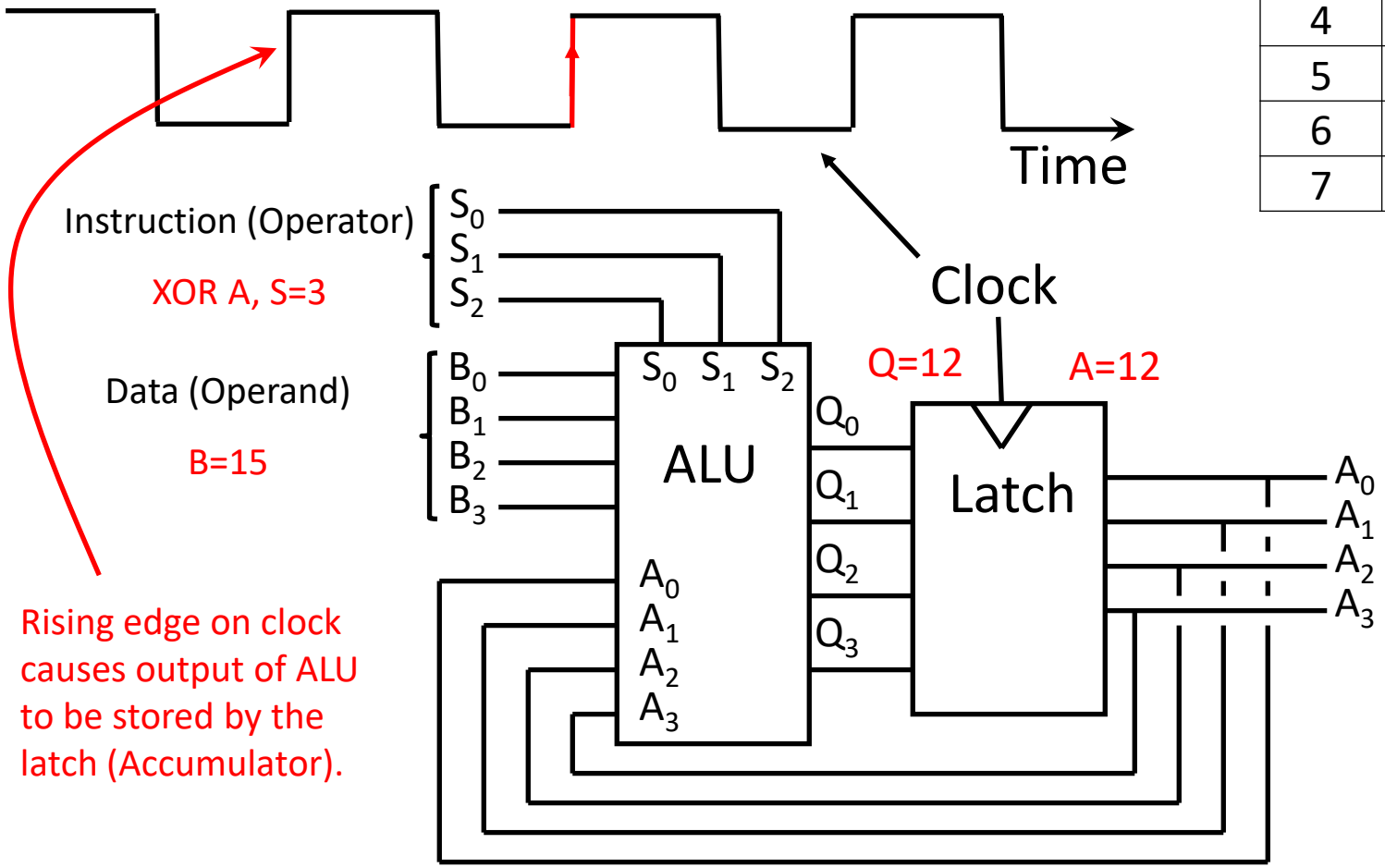


Program to evaluate 5-3

Assembly	Machine code	Accumulator
MOV A, 3	S=0, B=3	A=0011b=3
XOR A,15	S=3, B=15	A=1100b=12
ADD A,1	S=4, B=1	A=1101b=13=-3 TC
ADD A,5	S=4, B=5	A=0010b=2

ALU Instruction set

S(Hex)	S2	S1	S0	Function
0	0	0	0	MOV A,N
1	0	0	1	AND A,B
2	0	1	0	OR A,B
3	0	1	1	XOR A,B
4	1	0	0	ADD A,B
5	1	0	1	Shift R, A
6	1	1	0	Shift L, A
7	1	1	1	NOP



And so on…

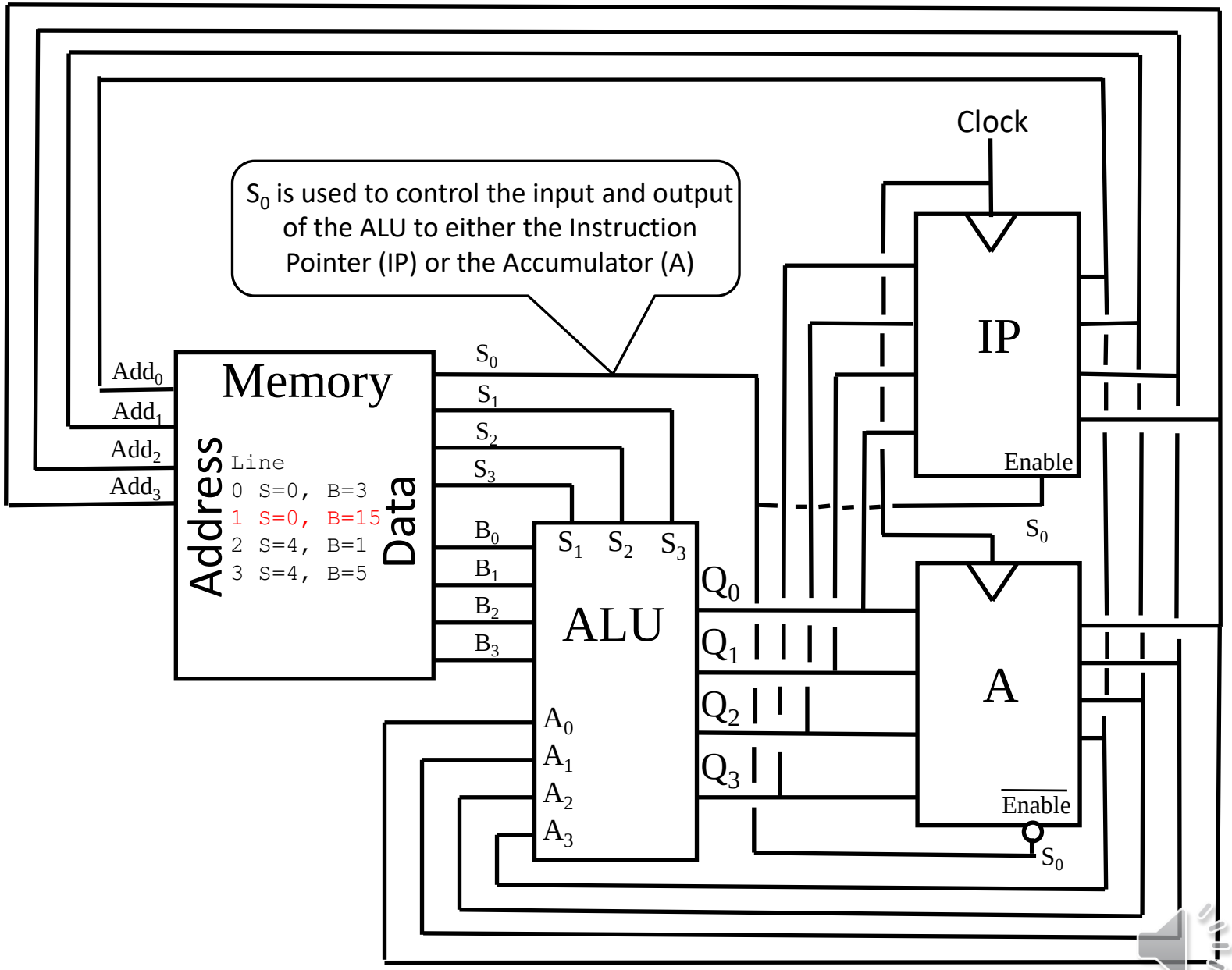
Adding memory and instruction pointer, IP, turns a calculator into a microprocessor

Adding a memory to store the program as machine code (S and B values) and a second register (IP instruction pointer) to store the address of the line code allows us to create a microprocessor.

Each clock pulse causes the IP to increase as to point to the next line of code.

Adding or subtracting to the IP value is equivalent to jumping forward or backward through the program.

The Accumulator (A) is used to build the answer. The IP keeps track of the line of code.



Circuit not complete, however it does convey the concept