CPU central processing unit of a computer. It is a synchronous unit (which is synchronized by an internal clock) It contains the ALU

Clock Rate: CR is the frequency that the internal clock runs at. It is measured in pulses or cycles per second.
CR is measures in Hz
eg. 60Hz means 60 cycles per second

Clock Cycle (CC) is the time length of one cycle, measured in seconds. Therefore
CR = 1/CC
CC can be measured in milliseconds (ms) = 1*10-3 seconds
microseconds (u) = 1*10-6 seconds
nanoseconds (n) = 1*10-9 seconds
For each of these CC measurements there is a corresponding Hz Measurement
1KHZ = 103Hz
1MHz - 106Hz
1GHz = 109Hz
Instruction Count (IC) Number of assembly commands that were performed by the program
Cycles Per Instruction (CPI) number of clock cycles required to perform the entire program divided by the instruction count, i.e. average cycles per instruction.
CPIi number of cycles required to perform one instruction of type i
CPU Time (Execution Time) Time it takes for the processor to execute a certain program , measured in seconds.
CPU-Time = Clock Cycle length in seconds * Number of Cycles in the progam or
CPU-Time = CC*CPI *IC = 1/CR * CPI * IC

$$MIPS = \frac{IC}{CPU\ TIME_{(sec)} x10^6} = \frac{IC}{CC\ x\ CPIxIC\ x10^6} = \frac{CR}{CPI\ x10^6}$$

$CPI = \frac{CPI_1 \cdot IC_1 + CPI_2 \cdot IC_2 + \dots}{IC}$

total cycles = CPI · IC

Amdahl's Law (Weinman's explanation):

$$ExTime_{new} = ExTime_{old} \times \left[(1 - Fraction) + \frac{Fraction}{Speedup}\right]$$

Amdahl's Law

$S_{latency}$ = 1/((1 - p) + p/s)
$S_{latency}$ is the theoretical speedup of the execution of the whole task
p (also called Fraction) is the proportion of execution **time** that the part benefiting from improved resources originally occupied.
s is the speedup of the part of the task that benefits from improved system resources

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - Fraction) + \frac{Fraction}{Speedup}}$$

SAMPLE:
15% -2.5 faster   20%- 3 faster   40%- 1.5 slower

$$ExTime_{new} = ExTime_{old} \times \left[(1 - (0.15 + 0.20 + 0.40)) + \left(0.15 * \frac{1}{2.5}\right) + \left(0.20 * \frac{1}{3}\right) + (0.40 * 1.5)\right] =$$

$$= ExTime_{old} \times \left[(1 - (0.15 + 0.20 + 0.40)) + \frac{0.15}{2.5} + \frac{0.20}{3} + 0.40 * 1.5\right] =$$

$$= ExTime_{old} \times 0.98$$

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{0.98} = 1.02$$

sign(1)	Exp (8)	Mantissa (23)

add 127 to exponent ↑ (IEEE 754 format)

Floating Point Arithmetic: raise the smaller power to match the bigger power

The algorithm for addition

Note: This will work for subtraction if we note that
A-B = A+(-B) = A+B'+1

```
 10        10
-01=>     +10
          +01
          ---
          101
       => 01
```

Given 2 floating point numbers:
1. Equate the powers by increasing the power of the smaller number and moving the decimal point accordingly
2. Add the numbers ignoring the power
3. Normalize the result into the form 1.bbbb * $2^{biased\ exponent}$
4. Check for overflow or underflow, if yes, create exception
5. Truncate (or round) the mantissa to 23 bits
6. If we get a negative result, we must make it positive and set the sign bit to 1
7. If there is a mathematical overflow, we place the overflow bit in the sign bit location

Multiplication

1. Write the numbers in the float format
2. add the exponents
3. multiply
4. normalize the result
   a. if the normalization created a new exponent, add it to the exponent from step 2
5. round the result if needed
6. put the point in the right place

Multiplication example: 5 * -0.4375

1. 5 = 1.010 *2^2, -0.4375 = -1.110 *2^{-2}
2. Sum of powers is 2-2=0
3. Multiply: 1.010 * 1.110 = 10.001100
4. Normalize: 10.00110 = 1.000110 *2^1
5. Round up: 1.001*2^1
6. put the sign on the number

```
     1.010
     1.110
  --------
     0,000
    10,100
   101,000
  1010,000
  --------
 10.001,100
```

Explanation of rounding: look at the digit after the desired precision. Add that digit to the last digit of precision.

Division

Same as multiplication except, in step 2 subtract the exponents instead of adding them

and in step 3 perform division

BOOTH:

Right most two bits of multiplier	action to be performed on the multiplicand
00	left shift 1
11	left shift 1
01	add to product and left shift 1
10	subtract from product and left shift 1

Note: For the first iteration, we look at only the single rightmost bit of the multiplier and assume a zero to its right, Thus, we either have 00 or 10.

Example with the booth algorithm for multiplication

We call X the multiplier (כופל) and Y the multiplicand (נכפל)
Let's say we want to multiply 6 * 3
So we have 0110 * 0011 (note that we add leading zeros so both numbers are the same length)
First let's make a product of twice as many bits 0000 0000
Now take the multiplier (the first number) and extend it to the size of the product by adding zeros on the left.
This gives us 0000 0110
Now we are ready to start the loop
1. We want to look at the bits $a_i a_{i-1}$ of the multiplier, so we need to shift it left one bit (using sll). This gives us 00001100. We only do this on first iteration.
2. Since the right two bits of the multiplier are 00 we are going to shift left the multiplicand (getting 00000110) and neither subtract nor add it to the product.
3. Now shift right the multiplier and examine the right two bits. They are 10, so we subtract the multiplicand from the product:
   0000 0000 - 0000 0110

Following rules of two's complement, we note that we can add the negative of the second number, giving:

```
   0000 0000
+  1111 1010 (remember: to make a negative value, flip all bits and add 1)
=  1111 1010
```

Or, in MIPS just use sub
Now, as in step 2. we left shift the multiplicand, obtaining:
0000 1100

Next we right shift the multiplier and examine the rightmost bits. These bits are 11 so we do nothing except left shift the multiplicand, obtaining: 0001 1000

Again we right shift the multiplier and examine the rightmost bits. They are 01 so we add the multiplicand to the product, as on right.

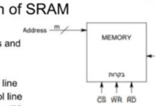
```
  1111 1010
  0001 1000
  ---------
+ 0001 0010 = 18
```

We could continue shifting the multiplicand left and the multiplier right but the multiplier has become all 0s so we will never perform any additions or subtractions. We are done. Also, i =n-1

RAM:
Sequential Access memory: we can only access one memory cell after the other. E.g. a tape drive. This means that accessing distant data will be slower.
Associative Access: This is a type of Random Access. We access by value instead of by address. We perform a parallel search.
Capacity: Maximum amount of data that can be stored in the system. E.g. 1TB 2.
Access Time: This parameter is defined differently for different memory systems.
<u>Random Access:</u> access time here as the time from when the address is supplied to the MAR until we can access it in the MIR.
<u>Non Random Access:</u> the time between the processor requesting the data from the controller and the controller supplying the data to the processor, not including the time it takes the controller to search the disk, just including the time to actually read the data.
Data Rate: Measured in BPS (Bits Per Second).
Cycle Time: Minimal time the processor must wait between the starts of two consecutive memory accesses.
Cycle Time = Access Time + Dead Time
Dead Time = Recovery Time + Transient Time
Transient Time: Time needed for the data to be electrically stable
Recovery Time: Time need to refresh the data, as necessary in DRAM

General Description of SRAM

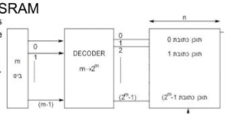
Memory size is 2^m*n where
2^m is the number of addresses and
n is the size of the data



RD (or OE) is the read control line
WR (or WE) is the write control line
When CS is 1 the data lines are TS
When CS is 0 the data lines are active
when CS=RD=0 and WR=1 then the value at address are read to the data lines
when CS=WR=0 and RD=1 then the data is written to the address in memory.

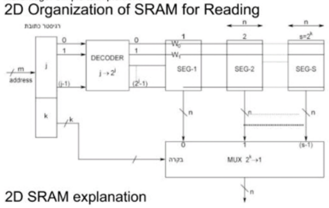
1D organization of SRAM

m is the memory space in bits
Using the decoder (מפענח) we select desired word (n bits).
The line out of the decoder is connected to an AND gate for that memory location.
This requires 2^m AND gates.



Disadvantage: Needs a lot of logic gates; for a tiny m=4 you would need 2^4 AND gates.
Advantage: Simple to implement.

2D Organization of SRAM for Reading



2D SRAM explanation

The Address is m bits divided into two sections j and k such that m=j+k

The Physical Memory is divided into s segments such that s = 2^k

Hence, the size of k determines the number of segments.

The width of each segment is the addressable word length, n bits (in mips, n=8).

The number of words (addresses) in each segment is 2^j. We note the number of segments, s.

We want a total of 2^m addresses in memory. s=2^m * 2^j and therefore s = 2^k.

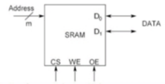
SRAM Memory Design

Goal: Enlarging the memory by using multiple small memories.

We will see methods for two types of memory expansion:

1) Increase the size of the data while keeping the address space constant.
2) Increase the number of addresses while keeping the data size constant.

Given SRAM of size 2^m X 2
OE=output enable
WE=write enable
CS=0 causes Data to be TS
CS=1 causes Data to be active

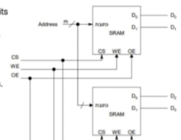


Increasing the Size of the Data (word length)

We will create a data with size 4 by combining 2 of the previous memory units

If we want to write 4 data we will put the data on lines $D_0D_1D_2D_3$ and set
WE=1 , OE=0 , CS=1
Then this data will be written to the address written on the address lines.

Read will be WE=0, OE=1, CS=1



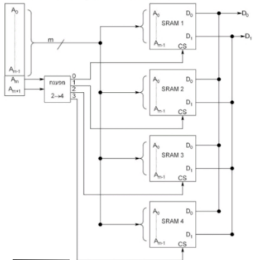
Expand memory by Expanding the Address Space

The data will remain 2 bits. We want to expand the address space by 4 and still have a 2 bit data.

The new size will be (2^mX4)x2 or $2^{(m+2)}$X2 . This means we need to add two bits to the address space.

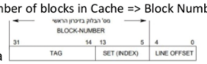
Note: In the following diagram for the sake of simplicity, we omit the WE and OE. But in reality they will be there.

Note: Since the memory output D_0D_1 are TS we are allowed to connect them to each other and only one will be active.

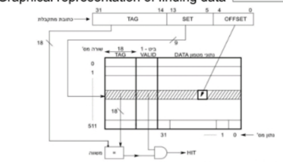


CACHE:
Hit: the datum is in the cache
Miss: the datum is not in the cache and must be brought from main memory
Hit-Rate: Number of access where the datum is in the cache/total accesses
Miss-Rate: Number of access where the datum is not in the cache/ total accesses
miss-Rate = 1- Hit-rate
Block: A unit of bytes by which we divide the Cache and Main Memory. Identical for both. If we need to bring a single byte into the cache we will need to copy the entire block in which it resides.
Hit-Time: The time it takes to either read or write a datum to or from the Cache.
Miss-Penalty: The time it takes to bring a block from Main Memory to the Cache, or to write it to Main Memory from the Cache. As we have said, when the processor reads from cache it reads one datum, but when we copy data into the cache from main memory, we copy an entire block.
Average Memory Access Time (for one datum) = Hit-Time + Miss-Rate x Miss-Penalty
<u>**3 Methods for Cache Mapping:**</u>
1. **Direct Mapped:** Simple to implement but not flexible
2. **Fully Associative:** Any block can be mapped to any location in cache. Flexible, high cache utilization, Complex to implement.
3. **n-way Set Associative:** Compromise between other two methods

Direct Mapping: Block Number Mod Number of blocks in Cache => Block Number in cache.
example address:

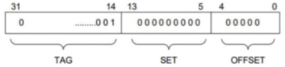


Graphical representation of finding data



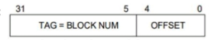
If data is found, HIT = 1

Write Through (WT): When the processor writes to cache it simultaneously writes to main memory. Note that only the single datum that was written gets written to main memory.
Write Back (WB): Only write to cache. When a block is removed from cache, if any data in that block was written to, the whole block must be written to main memory. We must add a **DIRTY** bit to the block. When data is written to a block in cache, that block's dirty bit is set to 1. Whenever a block in cache needs to be removed from cache, if the dirty bit is 1, the DMA writes the block back to main memory. Even for one data write to cache, the whole block will have to be written to memory.
Write Allocate: We make the appropriate space in cache available for the block. then we load the block from memory to the cache. Then we write the data following either WB or WT.
No Write Allocate: The processor updates the main memory directly without touching the cache.



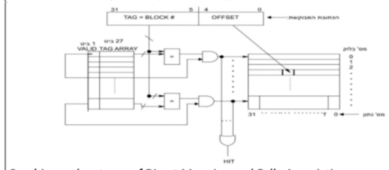
Set = the block number in the cache
Offset = the location of the data inside the block

Fully Associative Mapping:
NO SET FIELD



Finding the data in the cache by the address

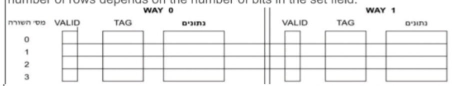
We get a memory address and want to see if it is in cache. We need to compare the Tag field in the address will all the tag fields in cache to see if the block is there. We do this comparison in parallel. Example of 32bit address with 5bit Offset



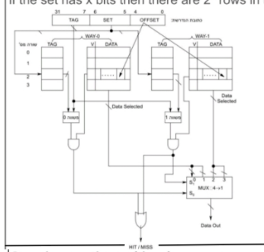
Combines advantages of Direct Mapping and Fully Associative.
Cache Row Number = (Block number in Memory) Mod (Number of rows in Cache)
<u>Note</u>: for the mod function we divide by the number of rows and not by the number of blocks and each row in cache has N blocks. See the diagram.

Cache Structure in 2-Way: Diagram

Cache memory is built like two tables where each table has a tag (ID) , the contents of the block (data), and a valid bit and a dirty bit if necessary. Note: the number of rows depends on the number of bits in the set field.



Explanation: the set field points to a row in the cache . That row has n blocks. In our case (2 way) - 2 blocks.
if the set has x bits then there are 2^x rows in the cache



Let us mark the valid bit by letter V.

The Set field of the address will select the row in the cache (eg. if the bits are 10, we choose row 2). We compare the Tag of the address to each of the two Tags in that row. We "and" the comparison with the valid bit, a 1 indicating a hit. (A matching Tag on an invalid block is not a hot, and will result in 0.)

The selected block in whichever "way" it resides, is sent to the multiplexer and according to the result of the comparison, the correct block will be selected.

Steps in executing multi-cycle processor:
1. **INSTRUCTION FETCH (IF)**
   A. The PC contents (which is an address of code) are transferred to the instruction memory.
   B. The instruction itself is read into memory.
   C. The instruction will be kept in a temporary register IR (INSTRUCTION REG)
   Symbolic representation of the operation: IR = MEMOR [PC]
   D. Increment the program counter to the next instruction: PC +4 → PC
2. **Instruction Decode (ID)**
   A. Retrieving the appropriate registers values from the Registers File (RF) to the ReadData1 port (this will be the contents of Rs) and the ReadData2 (This will be the contents of Rt). Let us denote: A = Reg (Rs) B = Reg (Rt)
   B. Decoding the instruction and creating control signals that will accompany the execution of the command
   C. We must Sign Extend the operand encoded in bits 0-15 (i.e. 16 bits) of the command itself even though we may not need this, just in case it turns out to be a branch command.
3. **Execute (EX)**

We now know what command we have. We perform the required actions on the A and B values that were calculated in step 2.
There are 4 options for this step based on the type of instruction:
A. R type instructions: The ALU performs an action on operands A and B, formally: RESULT-ALU = A op B
B. Accessing Memory (SW/LW): ALU calculates the effective memory address by adding the OFFSET to RS
C. Branch Commands: (Reminder: In step 2 the jump address was stored in the TARGET register) The ALU determines if A and B are equal by subtracting A - B. If the subtraction result is equal to 0 then flag Zero is set to 1 and the address in TARGET will be transferred to PC (i.e. a jump will be performed). If the result is different from 0, the PC register will remain with the value PC + 4.
D. Jump Commands: PC is to 4bits[PC+4] + 26bits INDEX << 2

4. (Only for Load, Store or R type)
<u>R-Type</u>: The result computed in step 3 which sits in the ALUOut register is written by the RF to register RD: Reg [IR (15-11)] = ALU Result
<u>SW:</u> The address computed in stage 3 gets sent to Write Address lines of the memory. Contents of Rt from the RF to Write Data of the memory. Control send the write command to memory: MEMORY = [ALU-Result] = B
<u>LW:</u> The address computed in stage 3 is sent to Read Address of memory unit, controller sends a read command to memory. The data read is put on the Memory Data lines of the memory Unit: Memory-Data = MEMORY[Alu-Result]
5. Write Back (Only for Load)
The memory Data retrieved in stage 4 is written to register Rt
Reg [IR (20-16)] = Memory Data

Control Line Settings

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-Format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

ALUOp (2-bits)
00 = add
01 = subtract
10 = R-Type

Five stages, one step per stage
1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

ALU Controller Outputs (Complete)

Instruction	Opcode	Function	ALU-op	ALU ctrl bits	Meaning
lw	35	-	00	010	add
sw	43	-	00	010	add
beq	4	-	01	110	sub
addu	0	33	10	010	add
sub	0	34	10	110	sub
and	0	36	10	000	and
or	0	37	10	001	or
slt	0	42	10	111	slt
add	0	32	10	010	add

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
lw $t4,8($t0)						IF	ID	EX	MEM	WB			
add $t5,$t1,$t4							IF	ID	EX	MEM	WB		
sw $t5,16($t0)								IF	ID	EX	MEM	WB	

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
lw $t4,8($t0)			IF	ID	EX	MEM	WB						
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
add $t5,$t1,$t4						IF	ID	EX	MEM	WB			
sw $t5,16($t0)							IF	ID	EX	MEM	WB		

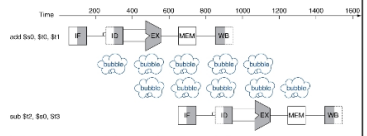
Structural Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

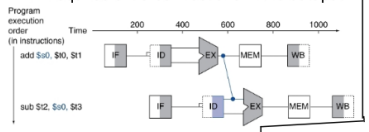
- An instruction depends on completion of data access by a previous instruction

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```



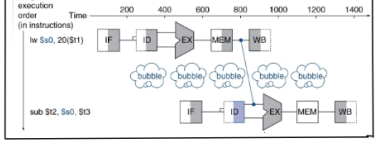
Forwarding (aka Bypassing)

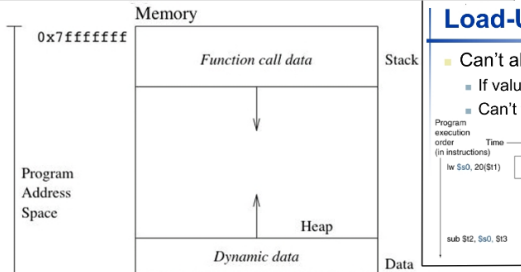
- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!





Binary Spot	Digit
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20
2097152	21
4194304	22
8388608	23
16777216	24
33554432	25
67108864	26
134217728	27
268435456	28
536870912	29
1073741824	30
2147483648	31

Cache Formulas

Block is composed of: Tag	Set	Offset

tag + set = logbase2(# of blocks in cache)

Size of main address − set − offset = tag

(size of main memory)/(size of block) = # of blocks in cache

2^{offset} = size of block

2^{set} = # of rows

N is for the n in n-way set associative (2-way SA -> = 2) $N*2^{set}$ = # of blocks in cache

Total bits in cache = (size of block * 8 + 2 + tag) * N * 2^{set} //in writeback

Total bits in cache = (size of block * 8 + 1 + tag) * N * 2^{set} //in writethrough

Stack = LIFO

Queue = FIFO

Miss − no load to cache

- Go to direct main memory o Write into memory directly

WB = Write Back

WA = Write Allocate

WT = Write Through

LRU = Least Recently Used

No WA = No-write allocate (also called write-no-allocate or write around): data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, data is loaded into the cache on read misses only.

1 Bit = 8 Bytes

4 Bytes = 1 Word = 32 Bits = 1 Instruction

2^{20} = megabyte