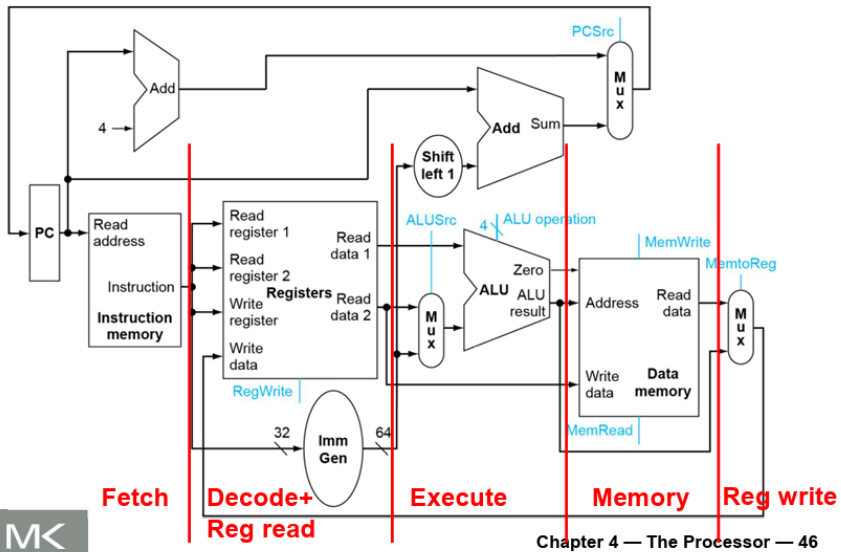
Multi-cycle Datapath



Five processing steps

IF: Instruction fetches from memory

ID: Instruction decode & register read

EX: Execute operation or calculate address

MEM: Access memory operand

WB: Write result back to register.

Logic design convention

Data path consist of two logical element

① combinational element → element operates on data value
② State element → element that operates on state

→ given the same input combinational element gives the same output as it has no internal storage.
ALU is example of combinational element given the input will give same output as it have no internal storage.

These element have storage because if we pull the power plug on the computer, we could restrat it accurately
ex:- Instruction, data memory, as well as register.

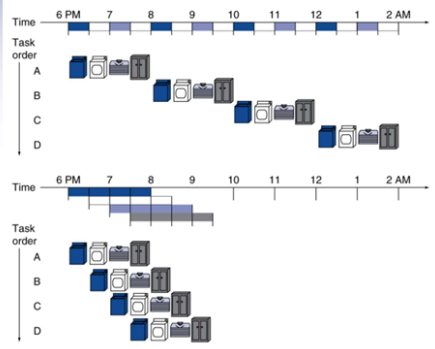
- A state element has atleast two inputs and one outputs.
  - Input
    - Data value to be written
    - clock → determine when the data value is written
- A state element can be read any time
- Logical component that contain state are also called Sequential as their output depends on both their inputs and the contents of internal state

Output of the state element provide the value that written in earlier clock cycle.

- RISC-V uses flip flops, memories and register as state element.
  D type flip flop is used by RISC-V which contains exactly two inputs (a value and clock) and one output.

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

Now see we have partitioned but we have not changed the time between rising edge of clock so it will be latency time and it will not improve time because 2nd instruction will not come until and unless the time does not change so we will use pipelining that will take the idea to divide 5 cycles and also change the time

So time required to execute next cycle will depend on the slowest cycle of the 5 cycles as you cant set the data until and unless each cycle has completed till then

See this as airport that have different counter .first people will go to counter 1 then it will go to counter 2 and counter 1 is vacant and another person can occupy that place similarly the line will be going on and after 5 counter finally the first person is done with everything and thus 1 instruction get completed.so until and unless next part is counter is clear no one previous to that is allowed. For simplicity we will take this max time as clock rising edge to favour simplicity.

A each cycle is of 0.5 and so total time is

=latency time + (no of stages -1)*time of one cycle =2 + 3*0.5= 3.5

Total number of cycle = p + (N-1) → Number of stages for larger p

↓ Number of instruction

Typical no of stages will be small only.

Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Here, max clock cycle = 200ps and thus it will decide latency time.

If you have few instruction say 1

① Time without pipelining → 200 + 100 + 200 + 200 + 100 = 800ps

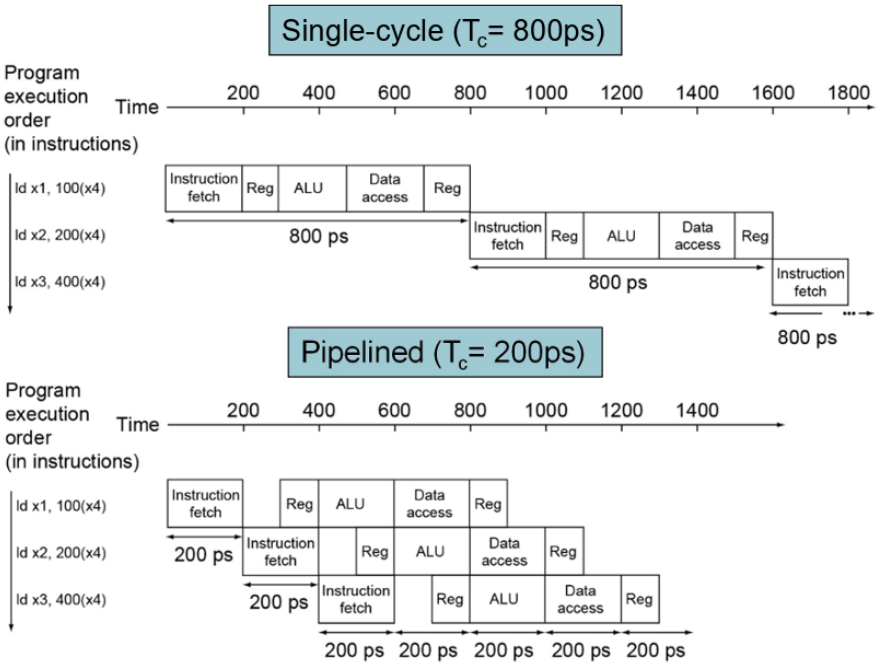
② time with pipelining

= so, each cycle will be 200ps (this is how we design)

= time = 200x5 = 1000 ps

so, for few instruction pipelining is not efficient but for large it is efficient.

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease even it will increase as we are chosing maximum cycle one and multiplying with no of stages

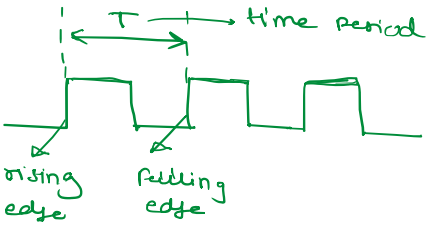
Clocking methodology

clocking methodology:-The approach used to determine when data are valid and stable relative to the clock. A clocking methodology defines when signals can be read and when they can be written. Stable and valid in the sense that one step (slowest step) has been properly completed to give its input that will be output of other

- A clocking methodology is defined to make hardware predictable. This also follow the principle that simplicity favour regularity. So, make the hardware to open & close at regular interval. even though we have to compromise somewhere.

Edge triggered clocking

In the state element input & output will come & go at the rising edge of the clock cycle.



Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements

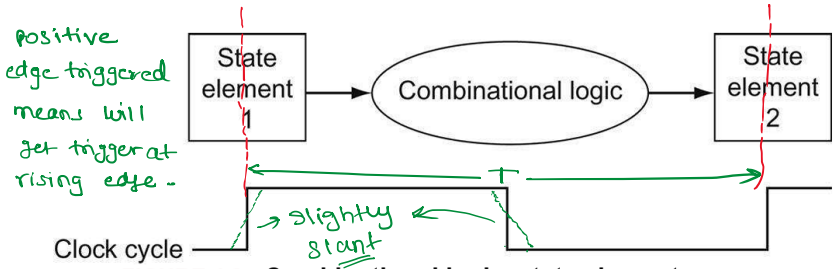


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related.

As combinational element is dependent on state or sequential element to get input and output. But state element will release and accept input & out at the edge of clock cycle. So, combinational element will get time period (T) to process the input + output ← So, design of time period will be based on slowest combination element.

control signal and write control signal

It is the signal will get generate when signal changes from low to high or high to low. It means at rising edge & falling edge.

But, we want to write/read data at rising edge. So, another control signal known as write control signal is used that will specify at which edge you have to write.

asserted :→ The signal is logically high or true.

Diasserted :→ the signal is logically low or false.

In 1 clock cycle ⇒ read the content of register of state element + procesing in combination element + write to the state element.

if a state element is not updated on every clock, then an explicit **write control signal** is required. Both the clock signal and the write control signal are inputs, and the state element is changed only.

See when we are processing some instructions then R/I type or load need to write back into register while store and branch does not.

If we not provide write control signal then it will unnecessary try to write the data in register that may cause the problem like it will try to write some where there is no register for write.so memory part which has write control signal will updated inly when both the condition are true. First is control signal is on ,2nd is data to be updated is stabilized and third is write /read control signal is on.

So write control signal will be valid at synchronized edge of clock signal.

What is mean by clock signal synchronization? —

See control signal(without synchronized) will get activated two times in a cycle…at falling edge and rising edge.so it is synchronized with one of them. Thus it generates signal only when that edge will come. In our cases rising edge is synchronized so data update will take at that point only.

Similar to “write control signal” read “control signal” also exist.

Note:- we generally not show control signal.

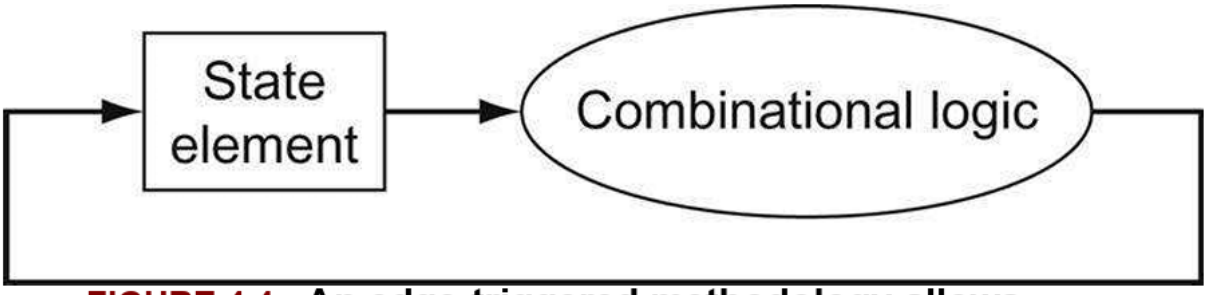


FIGURE 4.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values.

Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within one clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and on structures like the one shown in this figure.

Data buses will be 64 bit or 64 wire bus in 64-bit architecture. we may want to obtain a 64-bit bus by combining two 32-bit buses.

Datapath element: - A unit used to operate on or hold data within a processor. In the RISC-V implementation, the Datapath elements include the instruction and data memories, the register file, the ALU, and adders.

Operate on :- ALU, adders

Hold data:- data memories, register file ,instruction

program counter (PC) :- The register containing the address of the instruction in the program being executed

it is a register but a special register not general-purpose register. It uses adder that is a simplest form of ALU.

We can make adder by wiring the control lines so that the control always specifies an add operation.

True or false: Because the register file is both read and written on the same clock cycle, any RISC-V Datapath using edge-triggered writes must have more than one copy of the register file

False it is not necessary. see if there is one state element seen in figure in 4.4 at the edge of clock cycle it read data from register it processes then return the output to that register within time assuming the clock cycle is sufficient enough

no its should be right because you cant write the value untill and unless your register have been fully read. lets divide the input into parts suppose one part gets read then process but you cant store in the same register of state where the the reading is going on becasue it will corrupt your input as the other part of input is still in the process also combinational element not have storage device so you need an extra register to keep process values.

Actual Answer: - False. In a RISC-V Datapath with edge-triggered writes, you do not necessarily need more than one copy of the register file. The edge-triggered design means that the registers are written on the rising edge or falling edge of the clock signal, but this does not inherently require multiple copies of the register file. Instead, the design typically includes mechanisms to ensure that write operations do not conflict with read operations, such as using multiplexers or other control logic to manage the access to the register file. Multiple copies of the register file may be used in some designs for various reasons, but it's not a strict requirement for all edge-triggered RISC-V Datapath.

Feedback: - Feedback occurs when outputs of a system are routed back as inputs as part of a chain of cause-and-effect that forms a circuit or loop. The system can then be said to feed back into itself.

Instruction memory: - is a state element as it contains memory, but it is treated as combinational element.

The instruction memory need only provide read access because the Datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic because the output at any time reflects the contents of the location specified by the address input that is stored by PC.

Also, no read control signal is needed because every clock cycle one instruction will come and thus easily controlled by clock signal only.

Program counter: - This also a state element and uses two adder. It also does not uses any write control signal because in each clock it have to changed.

Register file: - A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed. It contains all 32 general purpose registers. out of which some are assigned for read and write and rest untouched.

The register file has 2 input known as register number to be read to read the register that is 5 wired bus while 2 output of 64 wired bus in RV-64.

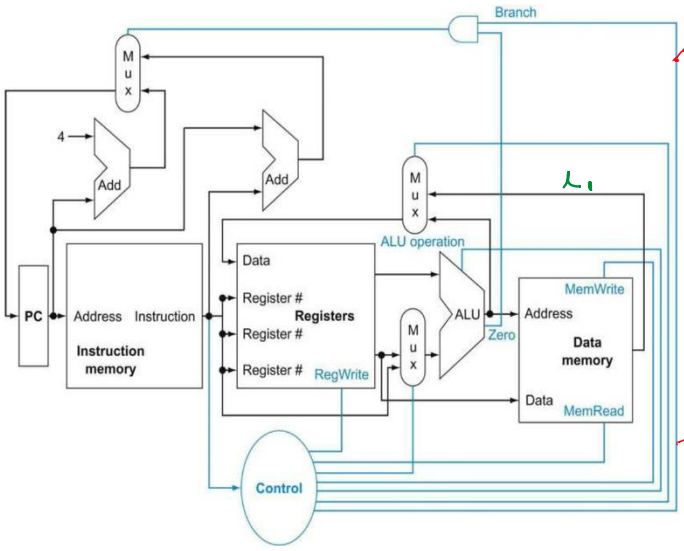
One input will there that will specify register to be written (5-bit bus) and one Data bus to written with 1 bit write control signal to decide write should take place or not in given cycle depending on instruction.

writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. (not much important)

ALU: - The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in Appendix A. We will use the Zero detection output of the ALU shortly to implement conditional branches.

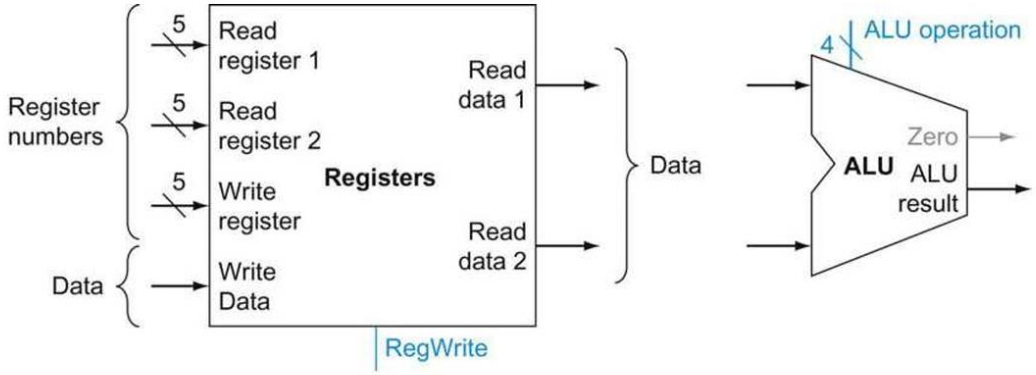
ALU only performs ADD, SUB, AND, OR but have assigned 4 bits instead of 2 bits. This is done for futuristic extension.

ALU also assigned a 1-bit signal if the result is 0. It is a zero-detector signal that do subtraction operation on two register coming and check 0.



① It is right time to answer your question that why we are using a control line for branch?
Ans → See zero detector is not type dependent. Either it is ALU, or Branch in both cases it will operate detect 0.
Suppose, SUB x_{10}, x_{11}, x_{15} You are doing. It will compare x_{11}, x_{15}. If they are equal then zero detection will send true. So, Branch control signal must be these to differentiate this is branch or not.

FIGURE 4.2 The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines.



a. Registers b. ALU

FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU.

The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section A.8 of

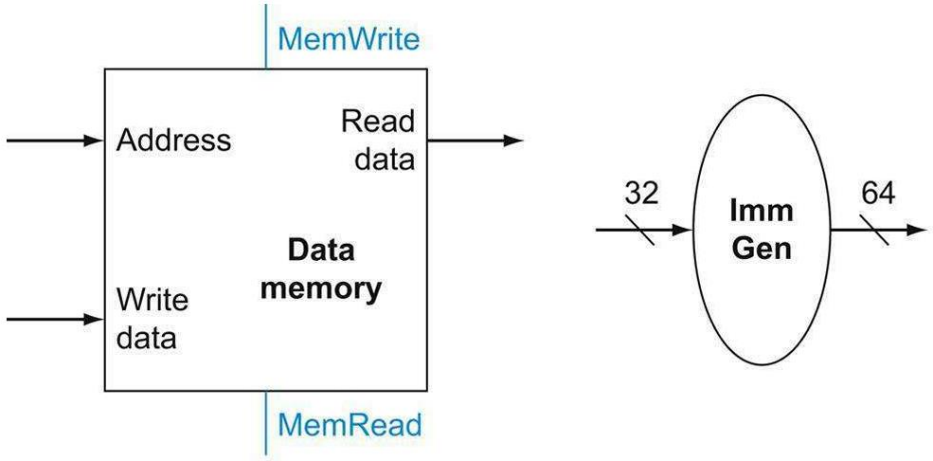
Let’s first come to your third point. I think any registers in pipeline stage can’t be solely write register or read register. The register which is being read must be getting write so it combines both.

Branch target address:- The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the RISC-V architecture, the branch target is given by the sum of the offset field of the instruction and the address of the branch.

In load and store instruction ld x1, offset(x2) or sd x1, offset(x2). And in immediate value offset will be of 12 bit but when we are doing ALU execution then all operation should be done on 64 bit as 64 bit output is there. So

There is an **Immediate generation unit** inside the ALU which will convert to 64 bit in RV-64.

Memory: -This is a state element **.**data memory has read and write control signals, an address input, and an input for the data to be written into memory.



See, we need memory write control signal because in load instruction we end up with writing the value. ld x_3 $0(x_1)$. If no control write signal will be there then it may be possible that you write the content of x_3 into $0(x_1)$ memory address.

But, But ----

why we need memory read control signal. See only problem will come in store register As it will both read and write to the memory. It will go to the line 1 shown is upper figure and then mux will allow this or this is load/store. It will go to data of register. But we have "regwrite" in register section i.e actually 0bb. So, it will not able to write.

Then why we are providing a redundent read control signal in memory.

Ans:-

There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a **read signal**, since, unlike the register file, reading the **value of an invalid address can cause problems**, as we will see in Chapter 5.

The immediate generation unit or (ImmGen) has a 32-bit instruction as input that selects a 12-bit field for load, store, and branch if equal that is sign-extended into a 64-bit result appearing on the output .We assume the data memory is edge-triggered for writes. Standard memory chips have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section A.8 of Appendix A for further discussion of how real memory chips work

Processor first check for higher level. If it found, then is called Hit and If not found then is known as miss.

The lower level in the hierarchy is then accessed to retrieve the block containing the requested data.

Block (or line) The minimum unit of information that can be either present or not present in a cache. If data will share, then it will happen in block (it is seen as similar analogy as information comes in the form of Book not pages in library---as you are allowed to take whole book or not)

Similarly, either Block will get transfer or not.

Hit Rate /Hit ratio: - fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy.

Miss Rate/Miss ratio: - fraction of memory accesses not found in the upper level.

Hit time: - The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

miss penalty: - The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.

If hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest (and largest) level.

As you wanting any amount of data it is providing to you as high hit rate and getting illusion that you are accessing huge chunk of memory or lowest level even you are not doing so.

Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy

Cost/speed: - L1 cache > L2 cache > L3 cache

STORAGE :- L1 cache<L2 cache <L3 cache

Main memory is implemented from DRAM (dynamic random-access memory), while levels closer to the processor (caches) use SRAM (static random access memory). DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon.

L3 sometime uses combination of S ram and D ram.

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

Taking inflation as 8% the cost has been double till now so calculate accordingly heeeee…

CACHE

Cache was the name chosen to represent the level of the memory hierarchy between the processor and main memory in the first commercial computer to have this extra level.

Note: --. The memories in the Datapath in Chapter 4 are simply replaced by caches. In place of memory, we will perform load and store on cache and this cache will further talk to memory if needed.

If ONE block contains only one word you are essentially not taking the benefit of special locality but when a block contain more than one word then you are going to load a chunk of address.

-----Remember we are loading a block from the memory but not keeping at same index continuous address will not go to same index .

Direct map: - The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the address of the word in memory. This cache structure is called direct mapped, since each memory location is mapped directly to exactly one location in the cache.

As a index in cache contain more than one word then how we will get to know the desired one reach at that point or not.

So we add tag different from the bits that were used for indexing is used to differentiate at same index.

Tag A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

when a processor starts up, the cache does not have good data, and the tag fields will be meaningless. Even after executing many instructions, some of the cache entries may still be empty, as in Figure 5.7. Thus, we need to know that the tag should be ignored for such entries. The most common method is to add a valid bit to indicate whether an entry contains a valid address.

Valid Bit: - A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

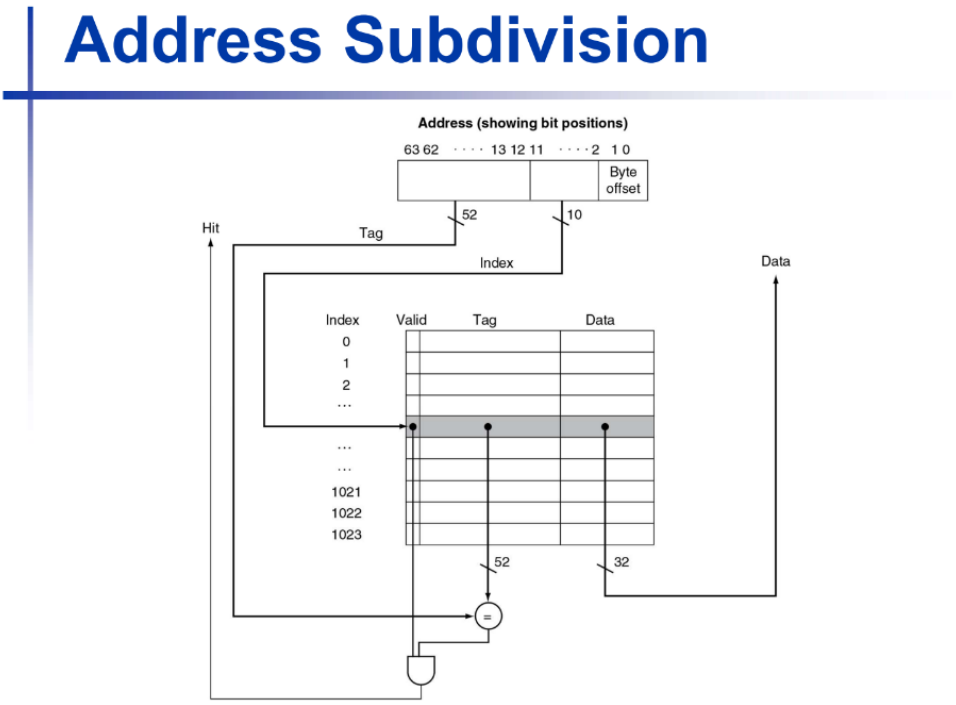
And for each data valid bit will be there or for each index one valid bit be there.

Okay if a block does not have associate then valid can be used to determine but if have associativity then for every data at common index should have valid bit? Right or not----

In general, handling reads is a little simpler than handling writes, since reads do not have to change the contents of the cache.

Dirty bit:- we add a bit (In virtual memory concept) to ensure that data should be written or not

Cashing: This is the prediction technique that and relies upon the principle of locality try to find the desired data in higher level of memory. The hit rates of the cache prediction on modern computers are often above 95%



This is direct mapping. It means what the cache will store as the address will point to that address in memory.

As memory address is generally 64 bits so you should generate 64-bit size from one

Now you have been given one block and each block have index associated it with as this is the property of the block.

Now we have to generate

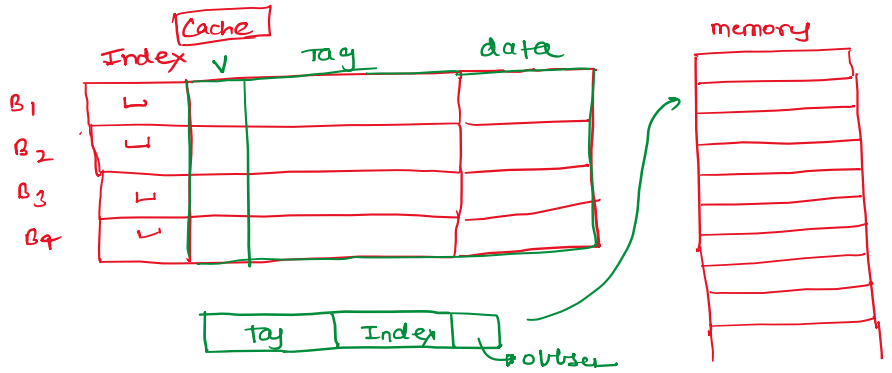
64-(index) bits to identify that address

It will also depends upon the number of word you are going to store in that block. We will see below. Lets n bits assigned for index it means 2^n index or blocks are there

Direct map

Block contain
1. 1 word.
2. more than 1 word.

Index is the basic property of cache that identifies the block of cache.

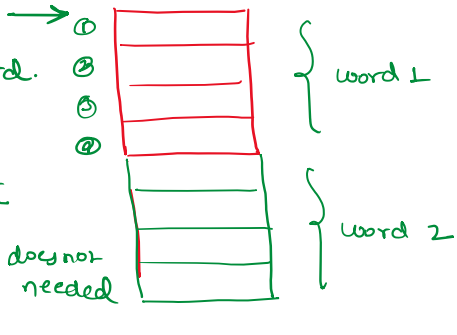


As this is direct map so tag + Index + offset = 64 bit and this 64 bit will map to memory and will put the values in data section of the block.

Q. Why offset came into play?

Ans → See, we generally load a "word" at least. Word contain 4 Byte and in Byte addressing we know one bit map to a Byte of the address.

If address is given of any of the 4 (①, ②, ③, ④) we can access the word. as we have seen.



So, 64 bit address 62 bit + 2 bit

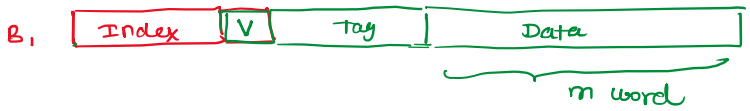
So if we able to to get 64 − (offset) = 64 − 2 = 62 ← 00, 01, 10, 11 does not needed

Internally we web will add 2 zeros at last and we can get that word in the memory.

Similary,

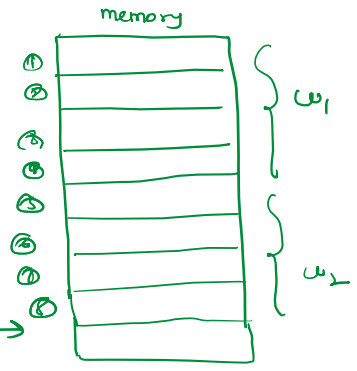
① If block contain more than one word say m words

what will be format of block then?



Note → we are loading m word as data. Let's take example of 2 word and try to relate.

As we are loading 2 word means 8 Byte (2×4 byte). So, values inbetween really should not matter as we got address ① we can load all Byte before ②.



& add. need is not there so we can ignore — — — (3 bits are of 64)

Offset = 3 bit

Similarly,

If want to load m word in block. offset will be $\log_2(4 \times m)$

$$\text{offset} = 2 + \log_2 m$$

Now tag = 64 − (index + offset) = 64 − (index + 2 + $\log_2 m$)

If m = 1 (1 word per block)

= 62 − index

block, uniquely specifies the memory address of the word contained in the cache block. Because the index field is used as an address to reference the cache, and because an n-bit field has 2^n values, the total number of entries in a direct-mapped cache must be a power of 2. Since words are aligned to multiples of four bytes, the least significant two bits of every address specify a byte within a word. Hence, if the words are aligned in memory, the least significant two bits can be ignored when selecting a word in the block. For this chapter, we'll assume that data are aligned in memory, and discuss how to handle unaligned cache accesses in an Elaboration.

The total number of bits needed for a cache is a function of the cache size and the address size, because the cache includes both the storage for the data and the tags. The size of the block above was one word (4 bytes), but normally it is several. For the following situation:

- 64-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks, so n bits are used for the index
- The block size is 2^m words (2^{m+2} bytes), so m bits are used for the word within the block, and two bits are used for the byte part of the address

The size of the tag field is

$$64 - (n + m + 2).$$

The total number of bits in a direct-mapped cache is

$$2^n \times (\text{block size} + \text{tag size} + \text{valid field size}).$$

Since the block size is 2^m words (2^{m+5} bits), and we need 1 bit for the valid field, the number of bits in such a cache is

$$2^n \times (2^m \times 32 + (64 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 63 - n - m).$$

Although this is the actual size in bits, the naming convention is to exclude the size of the tag and valid field and to count only the

size of the data. Thus, the cache in Figure 5.10 is called a 4 KiB cache.

Bits in a Cache

Example

How many total bits are required for a direct-mapped cache with 16 KiB of data and four-word blocks, assuming a 64-bit address?

Answer

We know that 16 KiB is 4096 (2^{12}) words. With a block size of four words (2^2), there are 1024 (2^{10}) blocks. Each block has 4 ×32 or 128 bits of data plus a tag, which is 64 −10 −2 −2 bits, plus a valid bit. Thus, the complete cache size is

$$2^{10} \times (4 \times 32 + (64 - 10 - 2 - 2) + 1) = 2^{10} \times 179 = 179 \text{ Kibibits}$$

or 22.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.4 times as many as needed just for the storage of the data.

- Given: Cache access time: 1 cycle
  Memory access time: 100 cycles
  Cache miss rate: 10%
- Find average memory access time
  - 1 + 100x0.1 = 11 cycles
- What should be the miss rate to have 1.5 cycles access time?
  - 1.5 = 1 + 100 x M
  - M = 0.005 (0.5%)

Example-2

- Address size = 32-bits
- Number of entries (4B each) in cache = 32
- Find number of bits for each field:

Tag	Index	Offset
25 bits	5 bits	2 bits

Example-3

- Address size = 40-bits
- Number of entries (8B each) in cache = 128
- Find number of bits for each field:

Tag	Index	Offset
30 bits	7 bits	3 bits

in

Essentially, Index of cache is the internal identity of index and thus not.

Thus, not counted in total bit of cache required.

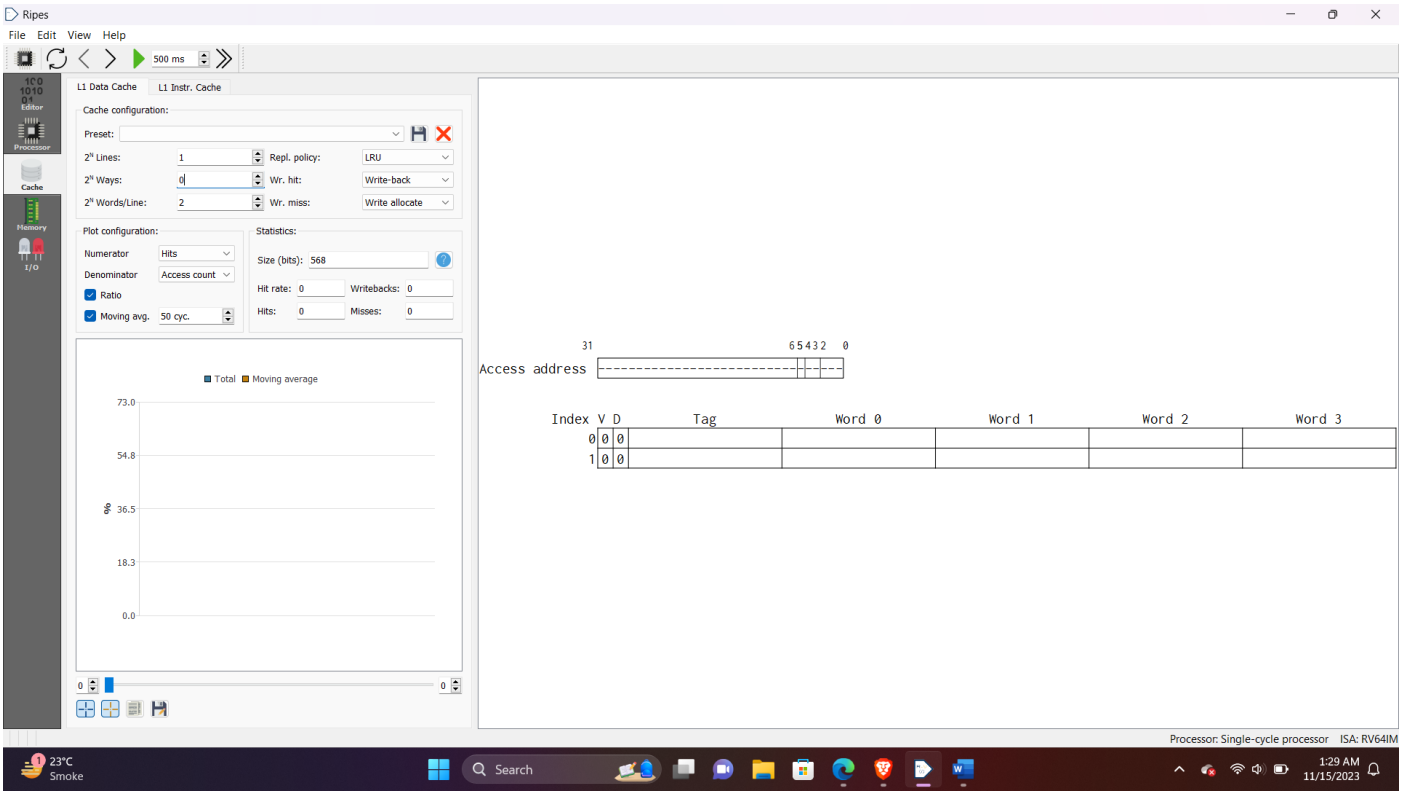
Total bits in cache is total number of blocks (data size+ tag size+ valid/dirty bit)

If we block size of cache is X byte it means data contained in block is of X byte and if the number of blocks in the cache is Y then it is known as XY byte cache.

Remember number bits in block = X*8 + tag bit + valid + dirty bit

Total bit in Block / complete cache size = no of block * (number of bits block)

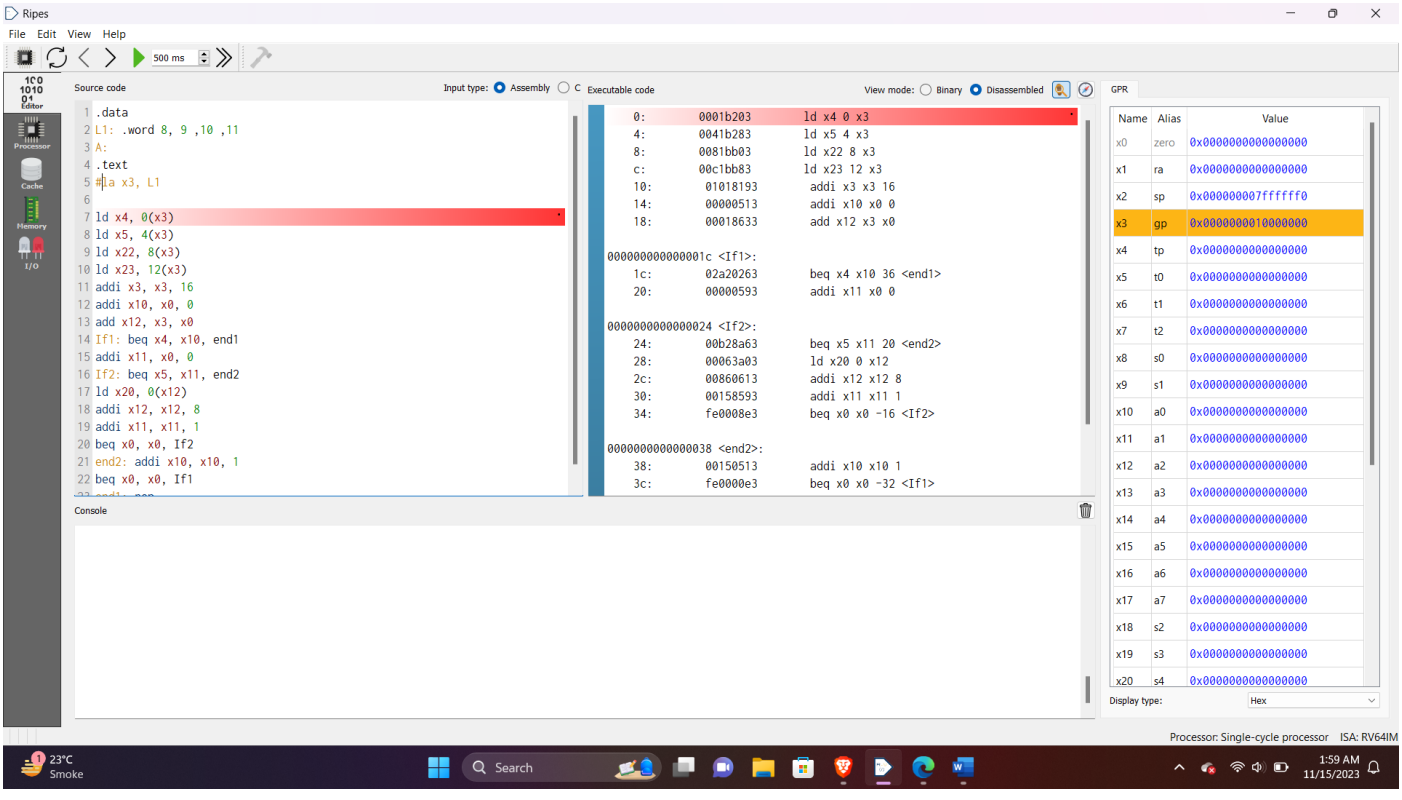
m m h (mhh mhh mhh)*8



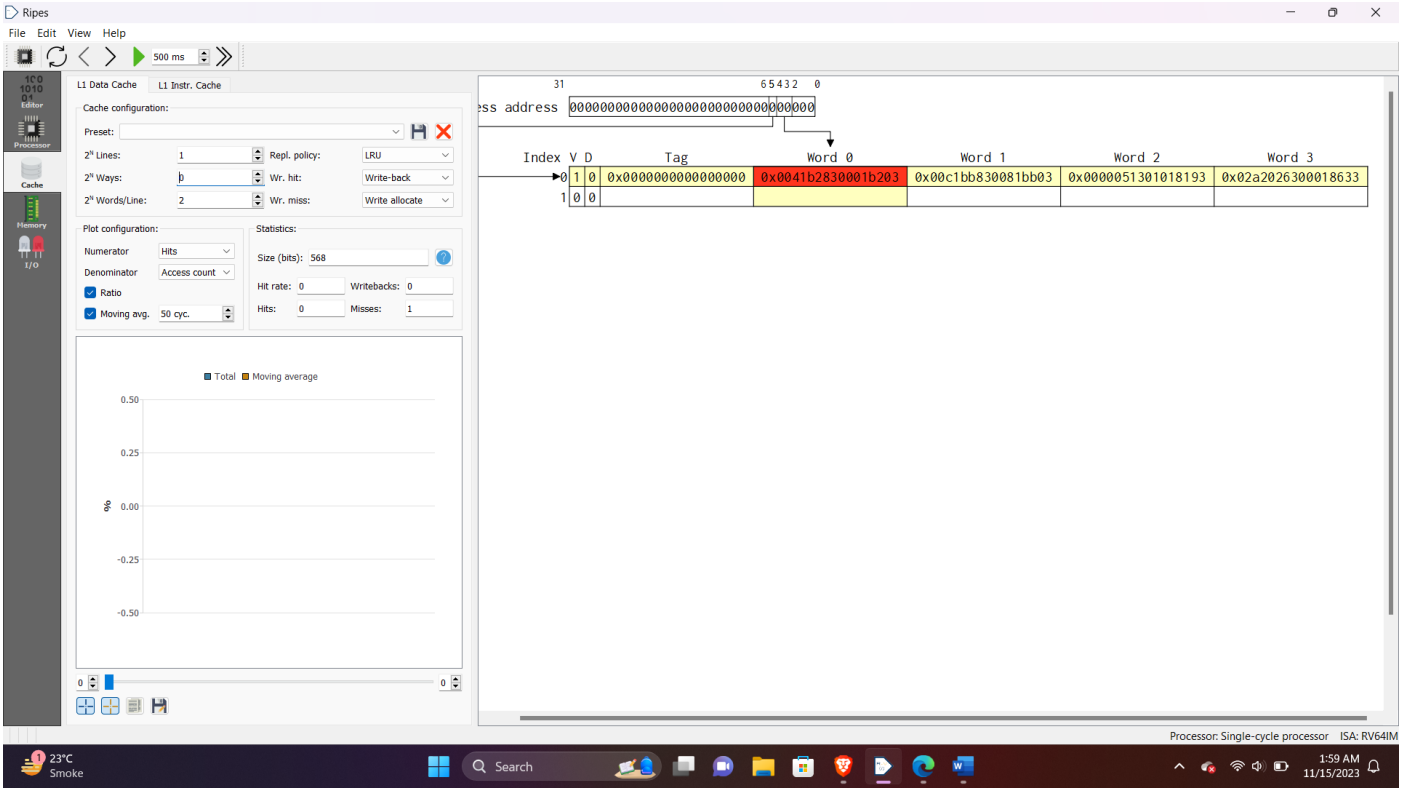
This is direct mapping as associativity is 2^0 =1

Word 0 /word 1 /word 2 /word 3 are double word not a word here in ripes. So when you are giving word/line = 2 it is taking 2^2 =4 double word in one block not 4 word. It have more capacity of loading. When you miss it will load next 32 byte data.

See in ripes simulator before clicking run arrow itself data will fetch and will will tell hit or miss after putting the arrow that instruction fully completed. So don’t get confused.



Corresponding cache below



Q.part 1

Program 1

(A) As this contain very specific type of address even the no of line is increasing but each one will have 1 miss and next 3 hit. If loading also wanting previous data then it must will get benefit. But in this program it is wanting next data at offset of 8 every time. So hit rate will be same.

Similarly for line 3 also same result will come and graph will be constant at approx. 75 %

(B) Blocks: 3, 4, 5 keping Lines as 3 and Ways as 0. In RIPES, Blocks refers to the number of blocks

Of course hit rate will increase due to increase in number of block as 2^3 –8 double word at a time hit rate will be 86 %

Now changing block will certainly increase the hit rate because at a time loading more size bloxk

Block 4 -16 double word at a time hit rate =92.43 %

Block 5-16 double word at a time hit rate =95.45 %

(c) (c) ways 0 block 2 and line 3

74.24 %

ways 1 block 2 and line 3

74.24 % No change because as index is same and each time it will anyway load the first one and rest 3 will will hit.

Program 2

(A) Low hit rate is due to discreteness of data . because we are adding every time x5 (slli x15, x5, 3) but x5 is 8 and noth changing it means that is nothing but slli 3 it means multiply by 8 but our one block contain essentially 4 double word at a time. X5 is 8 we want 8*8 the one or 64 one but we have present 8+8+8+8 = 32 so mostly time it will miss .Only two hit is there at the beginning it self .rest miss only. Asw block is two 4 double word we are loading at a time.

ld x20, 0(x12)

slli x15, x5, 3

add x12, x12, x15

Approx Same in all three--

Line 1 block as 2 -hit rate 3 %

Line 2 block as 2 -hit rate 3 %

Line 3 block as 2 -hit rate 4 %

(B) Blocks: 3, 4, 5 keeping Lines as 3 and Ways as 0. In RIPES, Blocks refers to the number of blocks.
Block is 3 so at a time it will load 8 double word or 8 *8-byte 64 byte and this is what we want. As we are using slli 3 it as seen above and ld wanting 64 byte each time and we have that.

Line as 3
Block 3- 86.36 %
Block 4- 92.42 % it will be higher of course as we are loading more number of blocks.
Block 5 – 95.45 same argument

(c) ways 0 block 2 and line 3

4.54 %

ways 1 block 2 and line 3

74.45 % due to associativity it will load 2 blocks will come at one index and effectively it it increases the space at given index and thus less replacement will take place so more data will be present there.

Q1.part 2

Program 1

Write-policies: Modify programs 1 and 2 to replace “ld x20, 0(x12)” to “sd x20, 0(x12)”. Use the preset cache configuration (32-entry 4-word direct mapped). Run program-1 and program-2 for various combinations of Write-through and Write-back policies (with allocate and without allocate)

As only read operation are there----

1.write back and write allocate - 74.24 %

Similar to ld we are keeping the sd and simple write back and write allocate is there so hit rate will be same as what was in ld

1.write through and write allocate - 74.24 %

Essentially both should be same as coming in hit rate only difference in write back and write through is how data is updated and time to update not the number of hit or miss as either it will be present in cache in both or not.

Write back and no write allocate: - 4.545 %

Write through and no write allocate: - 4.545 %

Same concept as using write back and write through in same policy will not change the hit rate.

When we are using no write allocate then the data is not going to cache and directly updating in memory and thus lefting the data un-updated so mostly it will get miss and thus have very low hit rate. Few hits come is beginning hit due to ld instruction otherwise nothing has come. Also point should be noted that

Program 2

Write through and write allocate: - 74.24 %

Write back and write allocate: - 74.24 % same concept as above.

Write back and no write allocate = 4.54 %

Write through and no write allocate = 4.54%

Same concept as the program as data is being updated in caches and directly getting updated in memory.so intital hit will come dud to ld and after that every time it will be miss only.

Q1 part 2 (for L 16 ,16)

Program -1

Write through and write allocate :-74.81 %

Write back and write allocate :-74.81 %

Value 16 is such that it is creating hit most of the time for program 1.Initially program miss will be more but it have loaded other memory block during that such that after some interval that will be present in the memory.

Write through and no write allocate :-1.163 %

Write back and no write allocate :-1.163 %

Same concept as the program as data is being updated in caches and directly getting updated in memory.so intital hit will come dud to ld and after that every time it will be miss only.

Program 2

Write through and write allocate :-1.163 % no of hits = 3 number of misses = 255 Total access = 3+ 255 =258

Write back and write allocate :-1.163 %

Write through and no write allocate :-1.163 %

Write back and no write allocate :-1.163 %

In this offset is such that it is not using the cache blocks that are loaded from memory and thus most of the time cache miss is taking place thus uses of cache is not very fruit full for the given associativity and lines and block size. Also use of slli in the program has increased the address needed more than it was present in a block and thus creating unavailability of data and thus very less hit rate.

Changing lines

1.cache read for program 2

Changing lines have no effect on the hit rate because its address is such that time it is accessing the data that is not present

Essentially it is requiring at lease 8*8 = 64 byte but at a time only 8*4 =32 byte can be load. Every time it wil

Question

Which block replacement policy is used for direct mapped cache?

A Optimal

B LRU

C Both (A) & (B)

D None of the above

Solution

The correct option is **D** None of the above
option (d)

In direct mapping at each index only one lock is stored in cache. Hence if new block come in cache then existing block should be replaced from the index. Hence no any choice of selecting one victim among multiple which mean no any replacement policy is required.

Number of blocks in main memory is 4M and number of blocks in cache memory is 4K If a cache tag directory entry contain 1 valid bit and 1 modified bit along with the address tag, then the size of cache tag directory (in KB) is _____for fully - associative cache mapping.

A 11

No worries! We've got your back. Try BYJU'S free classes today!

B 13

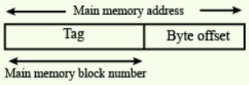
C 12

D 14

Solution

The correct option is **C** 12
option (b)

In fully associative mapping



If no. of blocks in main memory = 4M = 2^{22}

then main memory block number = $\log_2(2^{22})$ = 22 – bits

Find solutions

Cache tag directory size
= number of blocks in cache * 1 entry size

= 4K * 3B

=12KB

This is a good question. See memory may of some size maybe we don't know as it depends on the address size or number of bits in the address. But it will divide into blocks. See there are two types of blocks. One is block of memory and other is block of cache. Block in memory contain set of word. Here number of blocks in main memory is given and essentially this can be map by tag + index bits as no need to offset as block is there.

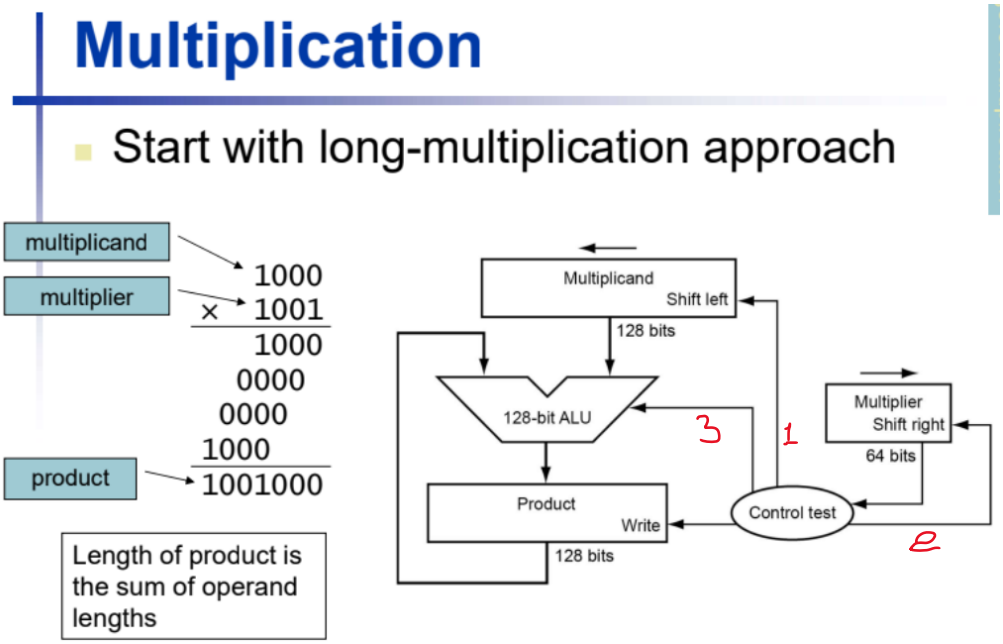
Now this is a fully associative cache it means only one map index in the cache so required index bit = 0

Tag+ index bit = tag + 0 = tag

Tag = log2(2^22) = 22 bits

So total bits in tag directory = 1+1+22 = 24 bits

And thus 3 Bytes.



If there are m bits and n bits operands, then maximum total number if bits after multiplication

= m + n.

Here 128-bit ALU is needed as we are using 2 64-bit operands.

The concept behind is if in 2nd bit, 1 is there then we are adding 1st operand else not adding 0(means not adding anything) also each time shifting by on place.

See what control test is doing---

First thing is storage. It is storing the value of multiplicand and multiplier. SO, line 1 and 2 are data bus that is providing multiplicand and multiplier to shifting unit while line 3 is control signal that is telling LSB after shifting multiplier that high or low signal. If high signal comes addition of multiplicand will be done as in the upper region as shown. If the control signal is 0 then internally it will add 0 instead of multiplicand. See a control signal is also going to product port that is deciding the register should be written or not. Suppose your MSB come 0 then you do not need to update product as ALU is bringing same value. You will conserve the writing effort. Also control test in line 3 tell adder inside alu that you have to work or not.

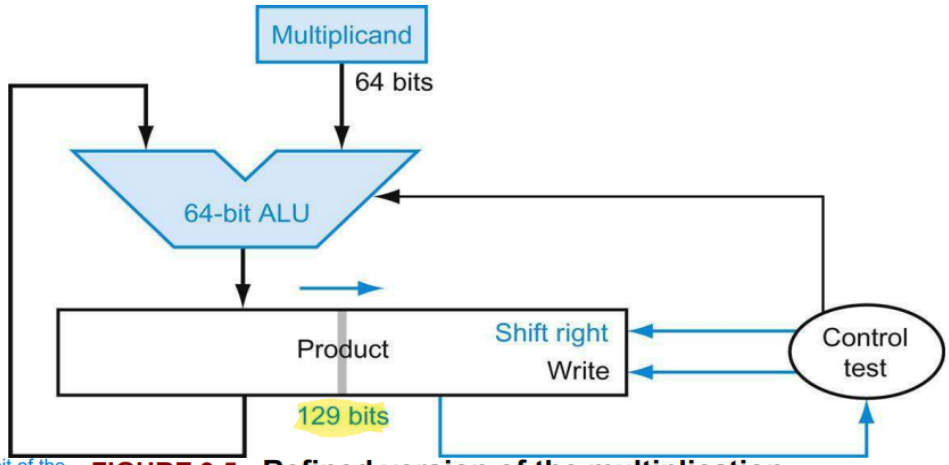
The Multiplicand register, ALU, and Product register are all 128 bits wide, with only the Multiplier register containing 64 bits. (Appendix A describes ALUs.) The 64-bit multiplicand starts in the right half of thevMultiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Over 64 steps, a 64-bit multiplicand would move 64 bits to the left. Hence, we need a 128-bit Multiplicand register, initialized with the 64-bit multiplicand in the right half and zero in the left half.

Figure 3.4 shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying with paper and pencil. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 64 times to obtain the product. If each step took a clock cycle, this algorithm would require almost 200 clock cycles to multiply two 64-bit numbers. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take several clock cycles without significantly affecting performance. However, Amdahl's Law (see Section 1.10) reminds us that even a moderate frequency for a slow operation can limit performance.

Improvised version of previous one

This algorithm and hardware are easily refined to take one clock cycle per step. The speed up comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a 1. The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand. The hardware is usually further optimized to halve the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.5 shows the revised hardware.



you are taking last 64 bit of the product
section every time as you can see from the
pencil multiplication that after shifting 1
last digit of product get fixed and will not affected further more.

FIGURE 3.5 Refined version of the multiplication hardware.

Compare with the first version in Figure 3.3. The Multiplicand register and ALU have been reduced to 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register, which has grown by one bit to 129 bits to hold the carry-out of the adder. These changes are highlighted in color.

because the addition of two 64bit operand can lead to at max 65 bits

but here point is you are providing 64 bit product and 64 bit operand in ALU but when overflows will happen in ALU itself then how you will get to know about carry-out. Shouldn't it be 65 bit alu. The 64-bit designation is more about the size of the data that the ALU can handle efficiently in a single operation not necessarily result.

Hardware/Software Interface

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left

note this is compiler dependent and every compiler essentially follow this to reduce the head

has the same effect as multiplying by a power of 2. As mentioned in Chapter 2, almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

Signed Multiplication

So far, we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember their original signs.

The first line means adding adder + adder + adder + ……

Such that one input of adder will be from outside while other will be from outside and same

Procedure continue 64 times.

2nd approach is there will be 64 .. n digit binary to be add so we will add two at a time with shifted ones and 32 result will come that also should be added using 16 adders with similar procedure.

A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 64 high. An alternative way to organize these 64 additions is in a parallel tree, as Figure 3.7 shows. Instead of waiting for 64 add times, we wait just the $\log_2(64)$ or six 64-bit add times.

As we are directly passing that carry to next operand instead of saving in product a and then again passing

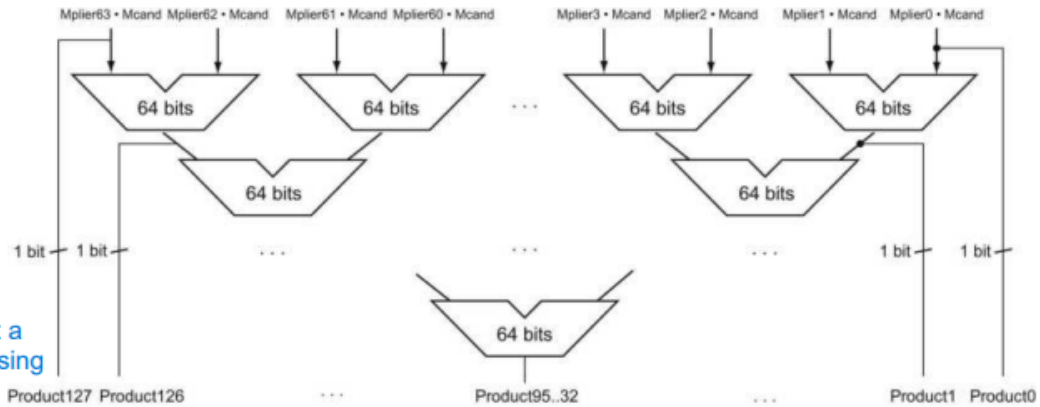


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 64-bit adder 63 times, this hardware "unrolls the loop" to use 63 adders and then organizes them to minimize delay.

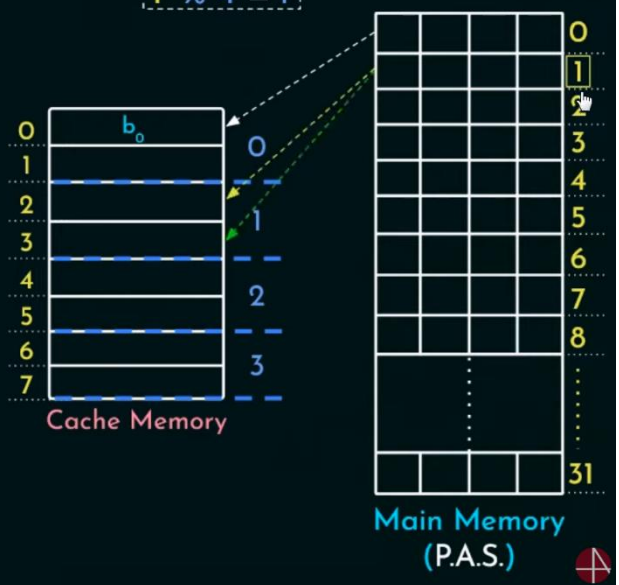
In fact, multiply can go even faster than six add times because of the use of *carry save adders* (see Section A.6 in Appendix A), and because it is easy to **pipeline** such a design to be able to support many multiplies simultaneously (see Chapter 4).

Ex : Byte addressable MM Size: 128 B
Cache Size: 32 B
Block Size: 4 B
2-way Set Associative | 1 set = 2 lines

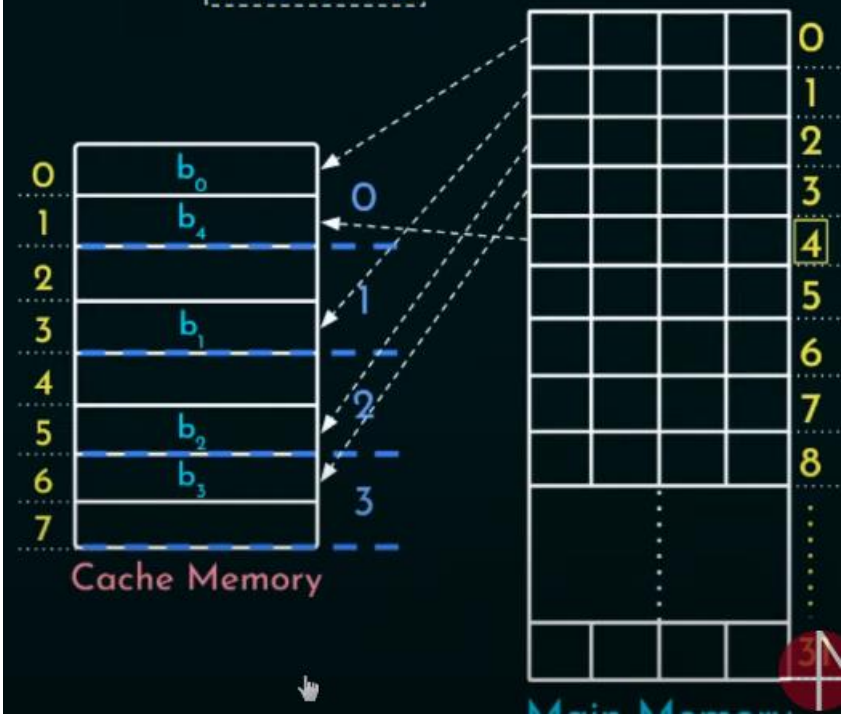
P.A.S. = 128 B = 2^7 B ∴ P.A. bits = 7
Block Size = 2^2 B ∴ offset = 2
∴ No. of MM Blocks = $2^7 / 2^2$
= 2^5
Cache Size = 2^5 B
∴ No. of lines = $2^5 / 2^2 = 2^3$
∴ No. of sets = $2^3 / 2^1 = 2^2$

7 - (2+2) = | 3 bits (Tag) | 2 bits (Set no.) | 2 bits (B/L offset) |
7 bits
5 bits

1 % 4 = 1



4 % 4 = 0



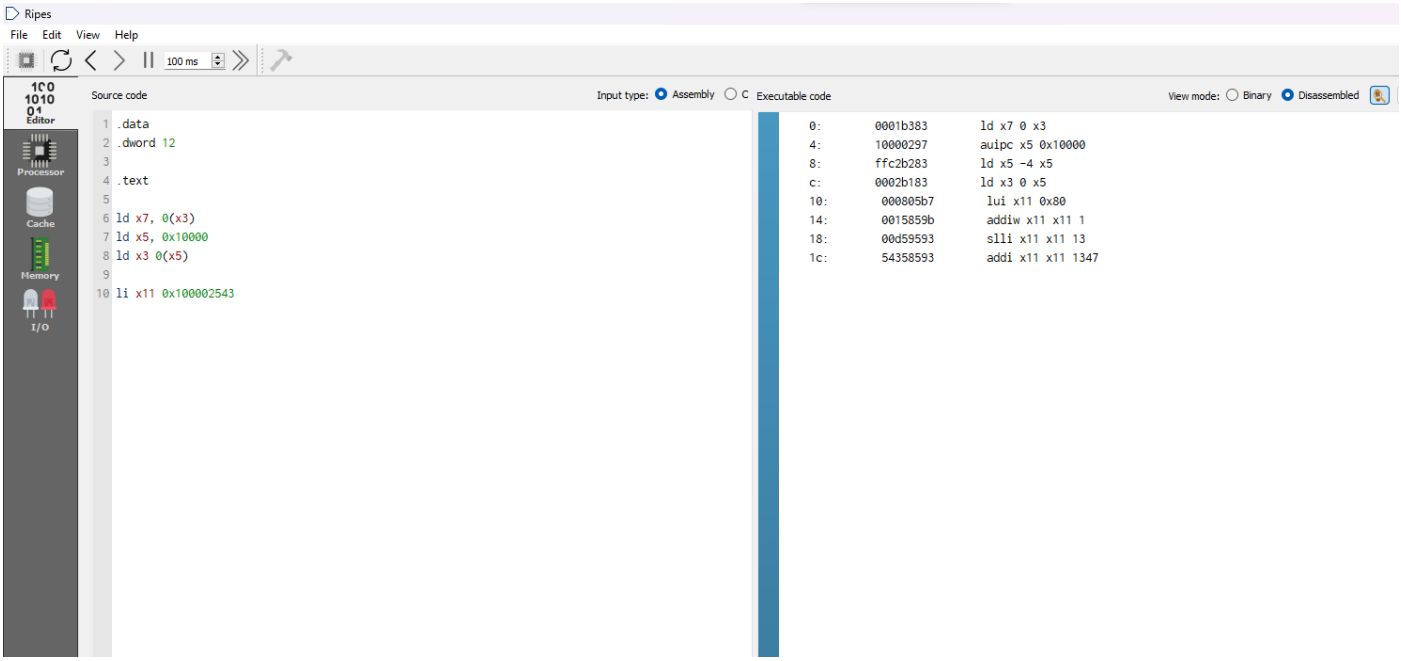


Table of Complexity Comparison:

Name	Best Case	Average Case	Worst Case	Memory	Stable	Method Used
Quick Sort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning
Merge Sort	$nlogn$	$nlogn$	$nlogn$	n	Yes	Merging
Heap Sort	$nlogn$	$nlogn$	$nlogn$	1	No	Selection
Insertion Sort	n	n^2	n^2	1	Yes	Insertion
Tim Sort	n	$nlogn$	$nlogn$	n	Yes	Insertion & Merging
Selection Sort	n^2	n^2	n^2	1	No	Selection
Shell Sort	$nlogn$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion
Bubble Sort	n	n^2	n^2	1	Yes	Exchanging
Tree Sort	$nlogn$	$nlogn$	$nlogn$	n	Yes	Insertion
Cycle Sort	n^2	n^2	n^2	1	No	Selection
Strand Sort	n	n^2	n^2	n	Yes	Selection
Cocktail Shaker Sort	n	n^2	n^2	1	Yes	Exchanging
Comb Sort	$nlogn$	n^2	n^2	1	No	Exchanging
Gnome Sort	n	n^2	n^2	1	Yes	Exchanging
Odd–even Sort	n	n^2	n^2	1	Yes	Exchanging

Mipes register and name

RISC-V register and name

Number	Name	Comments
$0	$zero, $r0	Always zero
$1	$at	Reserved for assembler
$2, $3	$v0, $v1	First and second return values, respectively
$4, ..., $7	$a0, ..., $a3	First four arguments to functions
$8, ..., $15	$t0, ..., $t7	Temporary registers
$16, ..., $23	$s0, ..., $s7	Saved registers
$24, $25	$t8, $t9	More temporary registers
$26, $27	$k0, $k1	Reserved for kernel (operating system)
$28	$gp	Global pointer
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address

Registers

Register	ABI Name	Description	Saver
`x0`	`zero`	Zero constant	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5-x7`	`t0-t2`	Temporaries	Caller
`x8`	`s0 / fp`	Saved / frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-x11`	`a0-a1`	Fn args/return values	Caller
`x12-x17`	`a2-a7`	Fn args	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x28-x31`	`t3-t6`	Temporaries	Caller

- Using the IEEE 754 floating-point format, which has a special representation for positive and negative infinity: **0x7F800000** and **0xFF800000** respectively [2] [3]. However, this requires converting between integers and floats, which may cause precision loss or undefined behavior.
- Using a variable-length encoding scheme, such as appending a sentinel value (e.g. **0x00**) to the end of the hexadecimal digits, or using a prefix (e.g**0x**) to indicate the start of the hexadecimal digits [4] [5]. However, this may require extra space or processing to encode and decode the values.