__/__/____

Tut-1

Combinatorial is another name for combinational

① Combinational circuit : outputs depends only on inputs. (no memory elements.)

Seq. circuit : Memory elements. (output depends on history of inputs.)

② GNU assembler. ('as'), g++ Compiler.

③ Algo → enhanced Java lang hierarchy.

HLL —Java Compile→ Java byte code

As. lang ← Java virtual machine.

Mach lang (responsible for doing mach dependent things)

D.L

④ Floppy disks.

⑥ If parallel tasks then → octa core ; {depends on clockspeed, Power consump etc}

⑨ MIPS ⇒ microprocessor without interlocked pipeline stages.

⑩ Networking devices & Automotive devices.

Q.7.

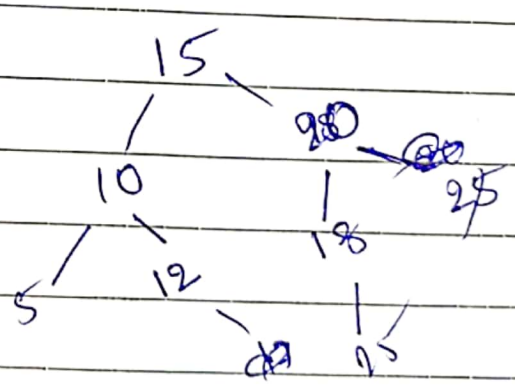
```
        addi  $s0, $zero, 100
FUNC:   addi  $s2, $zero, 0
        slt   $t0, $s2, $s0
        beq   $t0, $zero, EXIT
        sll   $t0, $s2, 4
        add   $s3  $t0, $s1
        lui   $t0, 1        (2^16)
        addi  $t0, $t0, 3   (2^16+3)  (65536+3)
        sw    0($s3), $t0   → x
        sw    4($s3), $t0   } a (2^32)(2^16+3)
        sw    8($s3), $0
        addi  $t0, $s2, 3
        sw    12($s3), $t0
        addi  $s2, $s2, 1
        j     FUNC
EXIT:
```

s3

char * y; // 32 bits.
};

align &M(i) → ptr
(16 bytes for each M)

Note big endian - higher order 4 bytes is at lower address



$S0 - save
$S1 → i
$S2 → k

Q.8

WHILE:

~~addi $s2, $zero, k~~

addi

muli $t0, $s1, 4 replace with sll $t0, $s1, 2

(a) add $s3, $t0, $s0

lw $t0, 0($s3)

beq $t0, $s2, NEXT

j EXIT

NEXT:

addi $s1, $s1, 1

j WHILE

EXIT:

(b)

bne $t0, $s2, EXIT

addi $s1, $s1, 1

j WHILE

EXIT:

This is the expected solution for (a)
For (b)
you need to implement like a do-while loop where condition is checked after body
However, since it is actually a while loop, you need to jump past the body straight to loop check (this jump will be executed only once)

(c) (a) = (b) + 1

(d) (a) → 70+6 (b) → 70+5

TUTORIAL 2

v0 does not have to be saved as caller, as v0 is a return value

Q1

Main	F	G
Callee : $ra, $s0, $s1	Callee : $ra, $s0, $s1	Callee : $ra, $s3
Caller : $a0	Caller : $a0, $t1	Caller : -

(Register Addressing)

Q2 ~~Opcode~~ will increase from 5 to 6 bits

R_c, R_t, R_d

Q3 $\frac{2^{15} - 1}{4}$ (space available) (1's 2 complement) $\Rightarrow 2^{13} - 1 \Rightarrow 8191$

(space req for int)

Q4 1 (a) Yes (p1.0, p2.0)

(b) Since $s1, $s0 are both callee preserved, # swapping them shouldn't cause an issue (so, YES)

(q1.0, q2.0)

2 a) Yes (p1.0, p2.0) (b) No, since we are using an external library, there callee & temporary) and $t1, $s0 are ~~[illegible]~~ and caller saved respectively.

t0 is caller saved and s0 is callee saved

Tutorial 3

1. a)
```
# Assume $a0 is a[0], $a1 is b[0]
# $t0 has i, $t1 has k $t2 has N, $s0 has c.
# $a2 is $a3 the max length of arrays 'a' and 'b'

    bge $a2, $a3, a3-lower
a2 lower:
    slli $a4, $zero, $a2, 2
    j a4-set
a3 lower:
    slli $a4, $zero, $a3, 2
a4-set:
    or $t0, $zero, $t1      # Set i = k
    slt $t0, $t1, 2         # i = 4k
loop-begin:
    slt $t3, $t0, $t2
    bne $t3, $zero, end-loop
    slt $t1, $t0, $a4
    beq $t4, $zero, end-loop

    addi $a2, $a2, 4
    addi $a3, $a3, 4
end-loop:
```

```
    sll $t3, $t2, 2
    add $a2, $a0, $t0
    add $a3, $a1, $t0
    j loop-begin
```

```
lw  $t5, 0($a2)
add $t5, $t5, $s0
sw  $t5, 0($a2)
```

b) Optimization already performed is the code above.

c) I can increment $a2 and $a3 by 4 after loading and storing to prepare for the next loop iteration

d) Both can be used, getting rid of the addi instructions in the end

2. a) or, sll, add, add, slt → 5 R formats

loop → bge
slti
beq
2 memory ops
add x4
jump

⇒ 3 branch, 2 memory and 5 R format instrs

⇒ (5 R formats) + (3 branch + 2 memory + 5R) × (N-k)

b) Using lw-inc and sw-inc, we get 20 in the second set of brackets

c)

3. a) No: of clock cycles Taken = 10^{10} clock cycles -
Finally, we have 9×10^{10} clock cycles

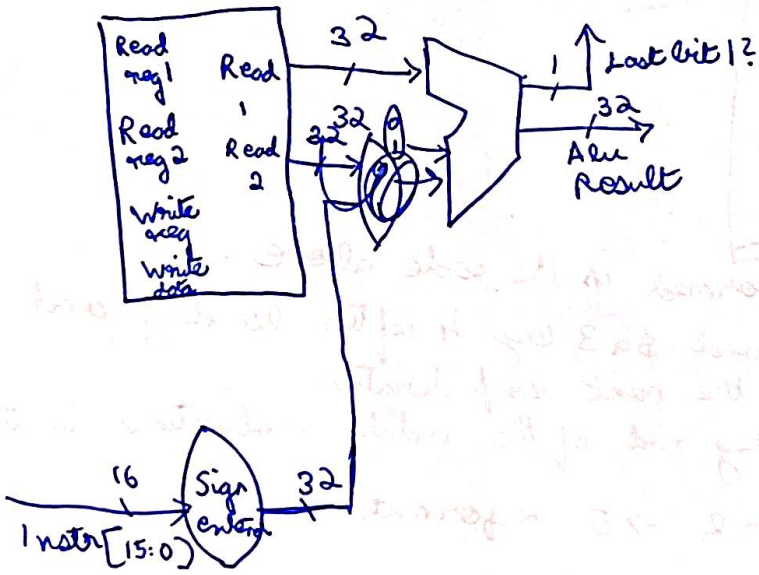
$\Rightarrow$ 5×10^8 mult operations were replaced.

b) Let every other line in the code correspond to a single clock cycle
$\Rightarrow 8 \times 10^9$ instructions. Now we have 9×10^9 instead of 8.5×10^9 instructions $\Rightarrow \frac{90}{85} = \frac{18}{17} \Rightarrow \frac{1}{17} : 1$

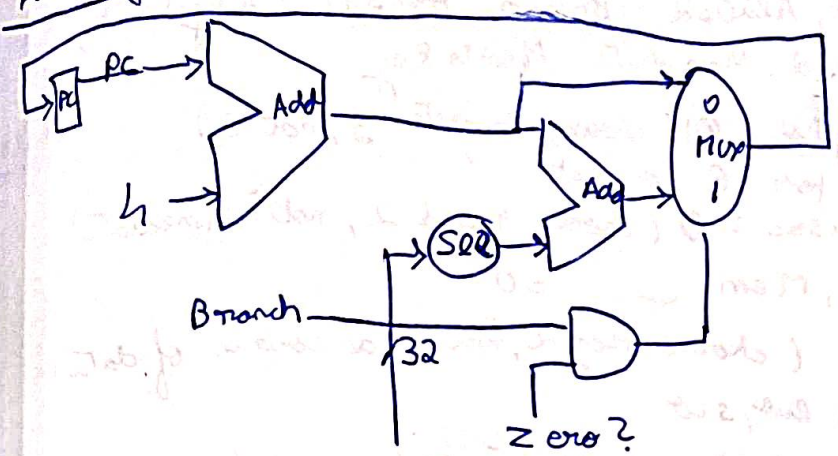
4. a) != is not defined between floats with infinite precision
Instead, consider the absolute value of (est × est − x) and bound it below, say, 10^{-5}.

b) Newton - Raphson iteration

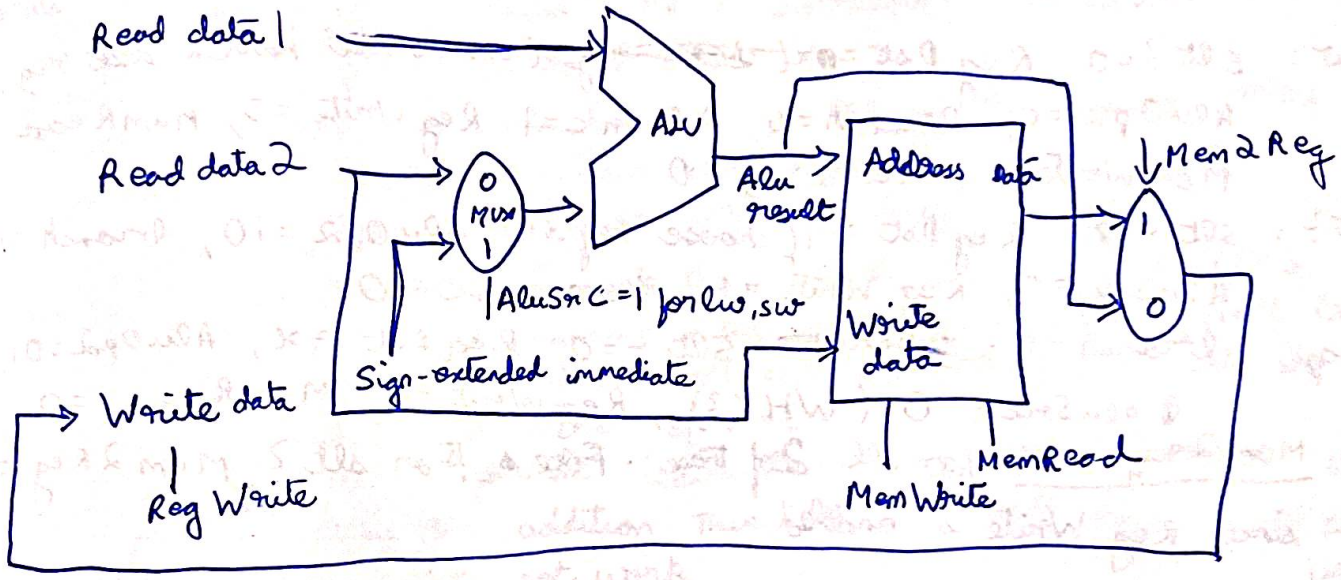
5.



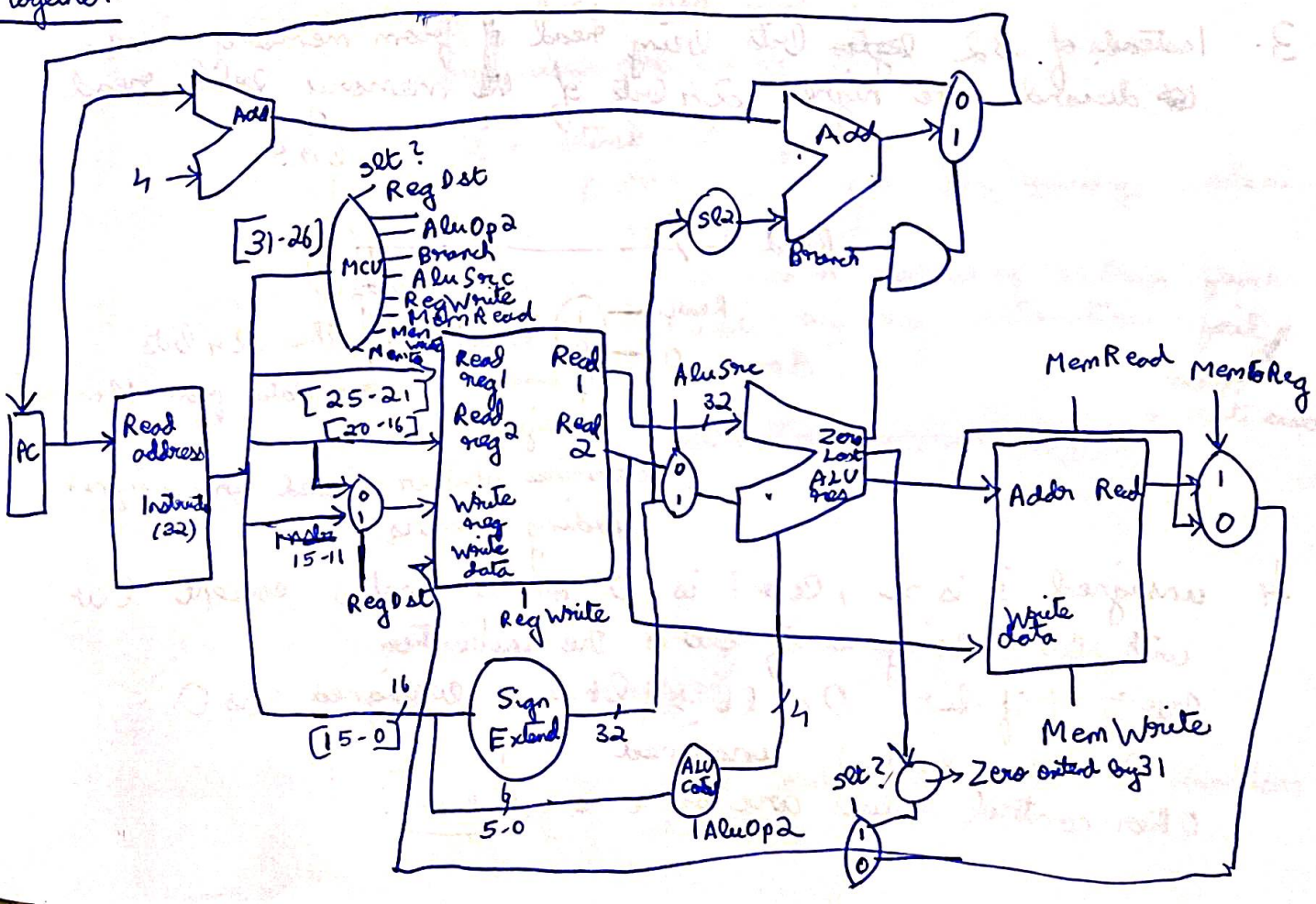
Handling beq:



Handling lw, sw:



Put together:

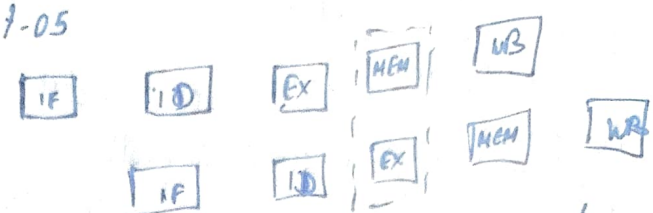


1) Jul-05

add3 r0, r1, r2

add3 r3, r4, r5

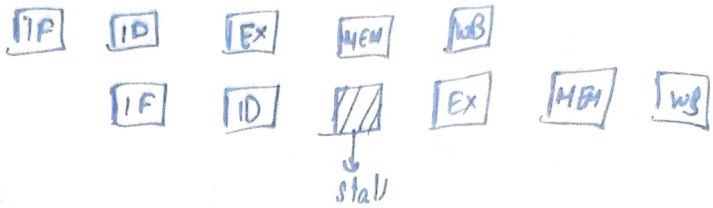
This solution is correct
- Bhaskar



Structural Hazard since same ALU

∴ add3 r0, r1, r2

add3 r3, r4, r5

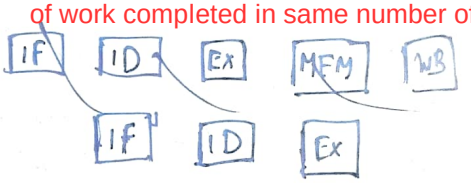


2) Yes, due to add3, considering only the structural stalls caused by it, we get atmost 1 structural stall due to add3, which matches with 2 instructions of (add, add) but the structural stall needn't occur always, add3 improves performance

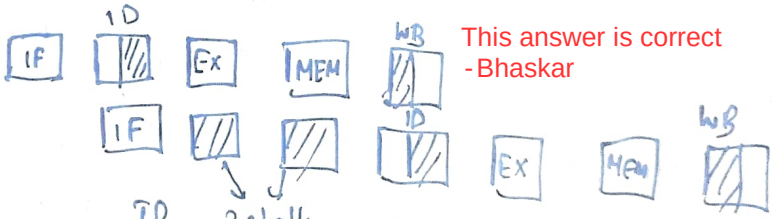
This answer is correct
- Bhaskar

If we make the simplifying assumption that add3 is followed by an instruction using ALU, then perf with/without add3 will be the same - same amount of work completed in same number of cycles

3. lw $t0 4($sp)
sw $t1 8($t0)



3. lw $t0 4($sp)
sw $t1 8($t0)



This answer is correct
- Bhaskar

5. a),b) ~~add~~ lw $sp 0($s0)

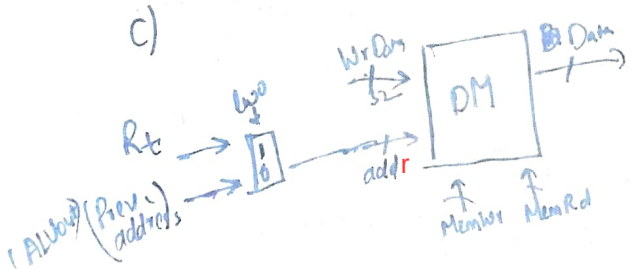
lw $t0 0($sp)

This ans is correct, except that usually we won't lw from memory to $sp
- Bhaskar

lw $sp 0($s0)

lw $t0 $sp

c)



This answer is correct
- Bhaskar

Forwarding to Ex stage

41 1) EX/MEM to EX

```
add $r1 r2 r3
add r5 r1 r4
```

2) MEM/WB to EX

```
add r1 r2 r3
nop
add r5 r1 r4
```

Forwarding to Mem Stage

3) MEM/WB to MEM

```
lw $t2 0($t1)
sw $t2 0($t3)   (with a stall)
```

4) Past WB to MEM

```
add R1, R2, R3
nop
Sw 0(R6), R1
```

Tut-6

Tut-6

1)

a) 3-stage branch completion
b) 2-stage branch completion
c) Assume branch not taken
d) Branch prediction
e) Branch target buffer
f) Delayed branches

Incompatible pairs = {a, b}, {c, d}, {c, e}, {c, f}, {d, f}, {e, f}

These two can be considered compatible. Branch predictor can return 3 possible values: 0, 1, don't-know
"Assume branch not taken" can be used when branch predictor does not know

similar argument

These 3 can be considered compatible in the case of
- 3-stage branch
- 1 delay slot only

Answer is correct with some explanatory clauses I have given
-Bhaskar

2)

(a) and $t1 $t0 $t2 — anything not producing s1 is ok

(b) and $s1 $t0 $t1 — anything producing s1 is ok

(c) ~~L8~~ - add $a0 $a3 $0 — anything not producing s2 or s3

(d) addi $a2 $a2 2 — this is not the cleanest answer - may be correct under some assumptions
some instruction producing s2 or s3 is the expected answer
e.g. sub $s2, $s0, $s1

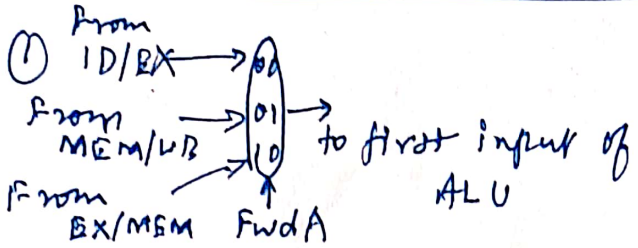
(e) addi $s2 $s1 1 — anything which does not read s2 is ok

(f) addi $s1 $s2 1 — anything which reads s2 is ok

(g) L1 - addi $s2 $s2 0
L7 addi $s2 $s2 3

answer should create a data hazard stall
- which means involving lw followed by a dependent instruction
L4 reads s2, make L7 also read s2 and make L1 load s2
e.g. L1: lw $s2, 4($sp)
L7: sub $s2, $s0, $s1

CS230 Tut 7 Solutions

(1) 

Start with same code as in slides (slide 7) - fwd from reg-reg to reg-reg

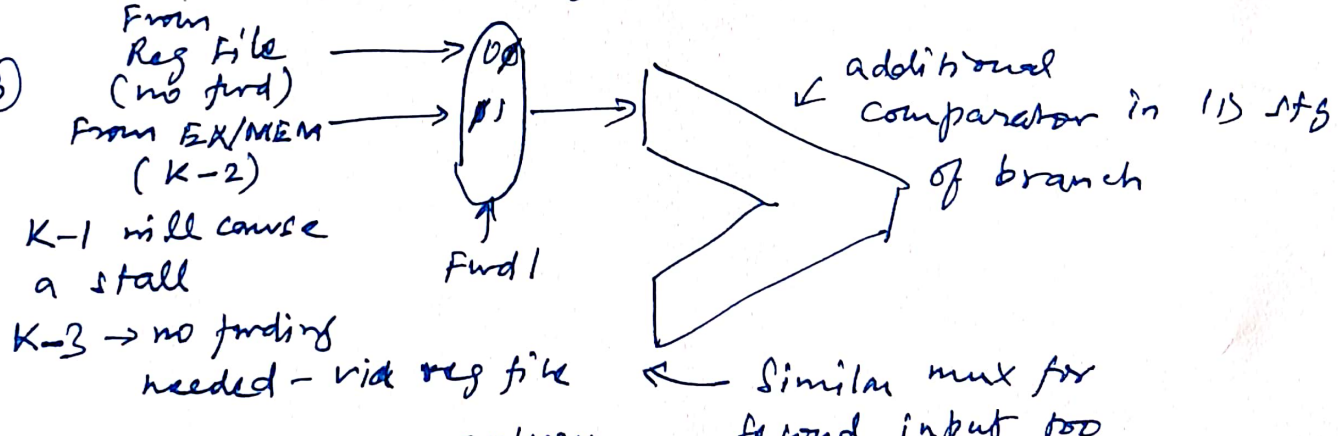
Add this condition at the end:

```
if((IF/ID.Rs == EX/MEM.Rt) &&
   (EX/MEM.MemRd == 1))
    FwdA = 01
```

Similar for FwdB - replace Rs with Rt

(2)
```
if ((IF/ID.Rs == ID/EX.Rt) &&
   (ID/EX.MemRd == 1))
    STALL
```

- can be combined with previous if conditions
- the STALL if case will be satisfied during the first time the dependent reg-reg instruction is in ID
- the forwarding if case will be satisfied during the 2nd time.

(3) 

(4)
```
if ((IF/ID.Rs == EX/MEM.Rd) &&
   (EX/MEM.RegWr == 1) &&
   (EX/MEM.MemRd == 0))
    Fwd1 = 1
else Fwd1 = 0
```

Similar for Fwd2, replace Rs with Rt

(5)
```
if ((IF/ID.Rs == ID/EX.Rd) &&
   (ID/EX.RegWr == 1) &&
   (ID/EX.MemRd == 0))
    STALL →
    IF/ID latch = 0
```

	CC1	CC2	CC3	CC4	CC5
lw	IF	ID	EX	MEM	WB
invalid opcode		IF	ID		

Invalid opcode triggers in CC3
In CC4, first instruction of invalid opcode exception handler is fetched. CC4 also detects misaligned mem exception which takes precedence

Q1. diff b/w registers and L1 cache as cache

- CPU can only directly access registers (as it is part of CPU memory) register memory is much smaller than faster access
- Data loaded in registers is dictated by program unlike L1 cache. This is the main/important/conceptual difference

Q2. Cache is SRAM, smaller, closer to CPU hence making it faster than main memory (DRAM)

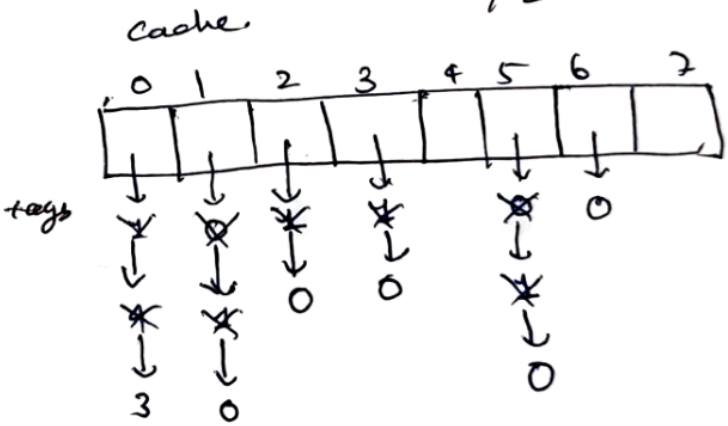
Q3. main memory = 2^8 words.
block size = 2 words
cache memory = 2^3 blocks.

last three bits of main mem block # → 7

word # n → main mem block # $\lfloor n/2 \rfloor$ —DM→ cache block # $\lfloor n/2 \rfloor \% 2^3$ → 8

tag → tag = $(n/2)/2^3 = n/2^4$

word #		cache block #	tag	hit / miss
2		1	0	miss
3		1	0	hit
11		5	0	miss
16		0	1	miss
21	10	2	1	miss
13	6	6	0	miss
64	32	0	4	miss
48	24	0	3	miss
19	9	1	1	miss
11	5	5	0	hit
3		1	0	miss
22	11	3	1	miss
4	2	2	0	miss
27	13	5	1	miss
6	3	3	0	miss
		6	0	miss



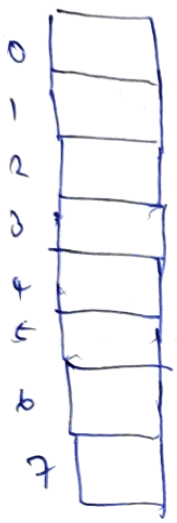
Q4 main memory size = 1024 words. = 2^{10} words

blocks in main memory = 2^9

blocks in cache = 2^3

would be same as Q3

since the seq, is same just tag size would be larger by 2 bits.



Q5. main memory = 32 MB = $2^{10} \cdot 2^{10} \cdot 2^5 = 2^{25}$ bytes

Cache size = 512 KB = $2^{10} \cdot 2^9 = 2^{19}$ bytes.

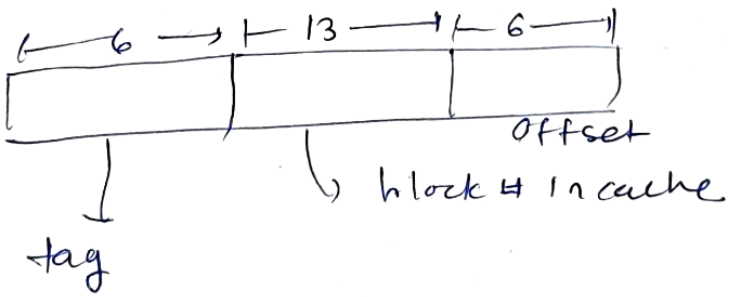
block size = 16-word = $2^4 \cdot 2^2 = 2^6$ bytes.
↳ offset

blocks in cache = $2^{19} / 2^6 = 2^{13}$

Assuming we need valid, dirty bits per block.

25 - (13 + 6)
6

DM:



0 bits comparator

metadata = (tag + 2) bits each block

= $2^{13} \cdot (8 + 2^3)$ bits.

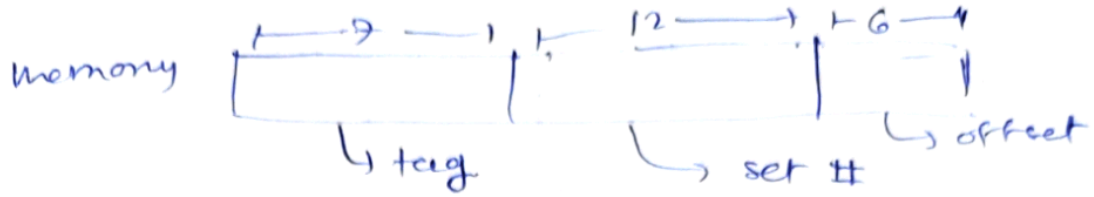
= 2^{16} bytes = 8 KB.

(b) 2 way set associative.

i) each set has 2 blocks

# of sets = $\frac{2^{13}}{2} = 2^{12}$

25 - (12+6)



ii) ~~1 bits each comparator~~

# of comparator = # of blocks in set = 2

iii) # of bits compared / comparator = tag size = 7

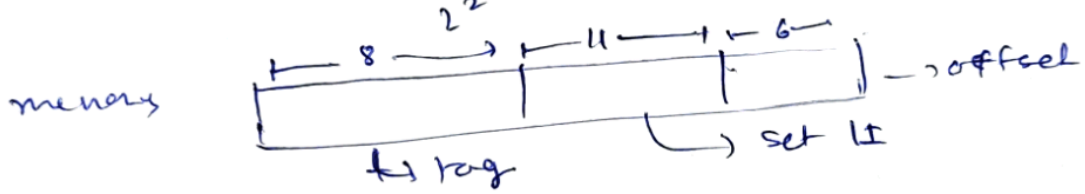
iv) metadata = 9×2^{13} bits = 9×2^{10} bytes = 9 KB

(c) 4 way set associative

each set has 4 blocks.

# of sets = $\frac{2^{13}}{2^2} = 2^{11}$

i)



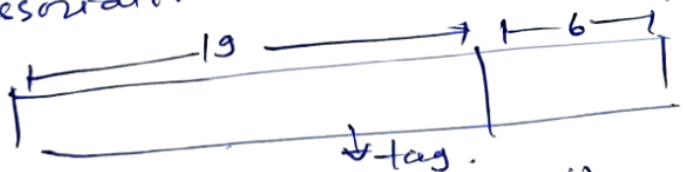
ii) # of comparator = 4

iii) # of bits compared / comparator = 8

iv) metadata size = $(8+2) \times 2^{13}$ bits = 10×2^{10} bytes = 10 KB.

(d) fully associative # of sets = 1

i)



ii) # of ~~bits~~ comparators = 2^{13} = # of blocks in cache.

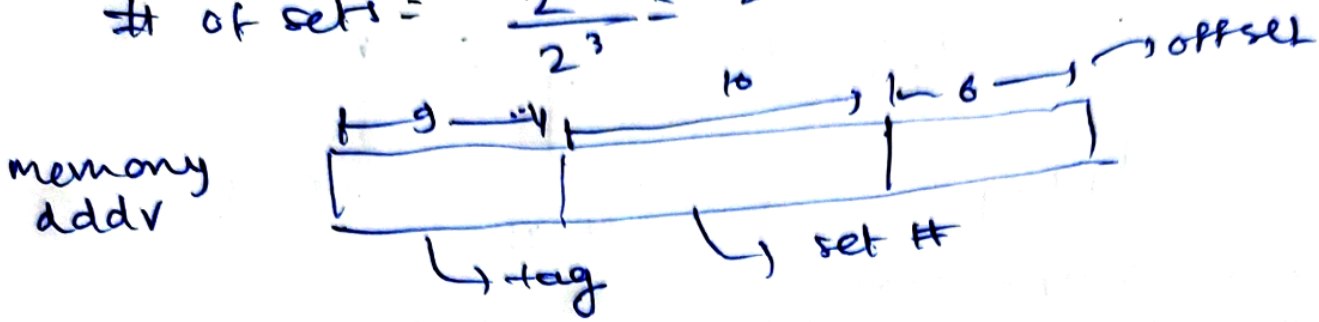
iii) # of bits compared / comparator = 19

iv) metadata size = $(19+2) \times 2^{13}$ bits = 21 KB

(d) 8-way set associative

i) each set has 8 blocks

# of sets = $\frac{2^{13}}{2^3} = 2^{10}$



ii) # of comparators = 8

iii) # of bits comp / comparator = 9

iv) metadata size = $(9+2) \times 2^{13}$ bits

= 11 KB

Q1. block size = 16 words.
miss rate = 0.25 % = 1/400 = r
CPI w/o miss = 2.5

DRAM Latency : a : comm ador : 2
b : accers : 20
c : comm data : 2

assuming : no early start policy
X - wide mem ⇒ bus size = X ~~words~~
Granuality of access is X

w/o interleaving
$$CPI = 2.5 + r * \frac{blocksize}{X} * (a+b+c)$$

with interleaving
$$CPI = 2.5 + r * \left(a+b+c * \frac{blocksize}{X}\right)$$

(a) X = 1 w/o interleaving $CPI = 2.5 + \frac{1}{400} \cdot \frac{16}{1} \cdot (24)$
$= 2.5 + 0.96 = \underline{3.46}$

with inter: $CPI = 2.5 + \frac{1}{400}(22 + 16(2)) = 2.5 + 0.135$
$= \underline{2.635}$

(b) X = 2
w/o interleaving: $CPI = 2.5 + \frac{1}{400} \cdot 8 \cdot (24) = 2.5 + 0.48$
$= \underline{2.98}$

with inter : $CPI = 2.5 + \frac{1}{400}(22 + 8*2) = 2.5 + 0.095$
$= \underline{2.595}$

(c) X = 4 w/o int : $CPI = 2.5 + \frac{1}{400} \cdot 4 \cdot (24) = 2.5 + 0.24 =$
$= \underline{2.74}$

with int : $CPI = 2.5 + \frac{1}{400} \cdot (22 + 4*2) = 2.5 + 0.075$
$= \underline{2.575}$

(d) Y = 8 w/o int : $CPI = 2.5 + \frac{1}{400} \cdot 2 \cdot (24) = 2.5 + 0.12$
$= \underline{2.62}$

with int : $CPI = 2.5 + \frac{1}{400}(22 + 2*2) = 2.5 + 0.065$
$= \underline{2.565}$

Q2. Half of instr. have data access

processor	cache type	block size (word)	instr miss rate (imr)	data miss rate (dmr)
P_1	DM	1	4 %	6 %
P_2	DM	4	2 %	4 %
P_3	2-way set	4	2 %	3 %

cache miss penalty = 6 + block size (words)

measured CPI (P_i) = 2.0.

(a) measured CPI = ideal CPI + $\left(imr + \frac{dmr}{2}\right)$ * miss penalty.

for P1

$2 = \text{ideal CPI} + \frac{4}{100} \times 7 = \text{ideal} + 0.49$

ideal CPI = 1.51

(b).

P_2: miss penalty = 6 + 4 = 10

$imr + \frac{dmr}{2} = 4\%$

$CPI = 1.51 + \frac{4}{100} * 10 = 1.91$

for P_3

(c) not enough data to compute effect of associativity on hit time → CPI and overhead via block eviction policy

[→ assuming there to be negligible for this prob.

P_3: miss penalty = 6 + 4 = 10.

$imr + \frac{dmr}{2} = 3.5\%$.

$CPI = 1.51 + \frac{3.5}{100} \times 10 = 1.86$

. P_3 is the fastest.

Note: number of blocks in set must be 3 x 2^k so that number of sets in cache is a power of 2, and also number of words in block is a power of 2

Q3. Cache size = 3K

- to calculate offset ~~# of~~ blocksize must be power of 2

(a) for D.M. and 2-way set ~~#blocks/sets must be~~ # sets in cache is not power of 2 hence both not possible with simple bit extrations.

(b) fully associative max blocksize = 1K words

(c) blocksize = 1K words = $4 \times 2^{10} = 2^{12}$ bytes.

mem addr field