CS M151B:
Computer Systems Architecture

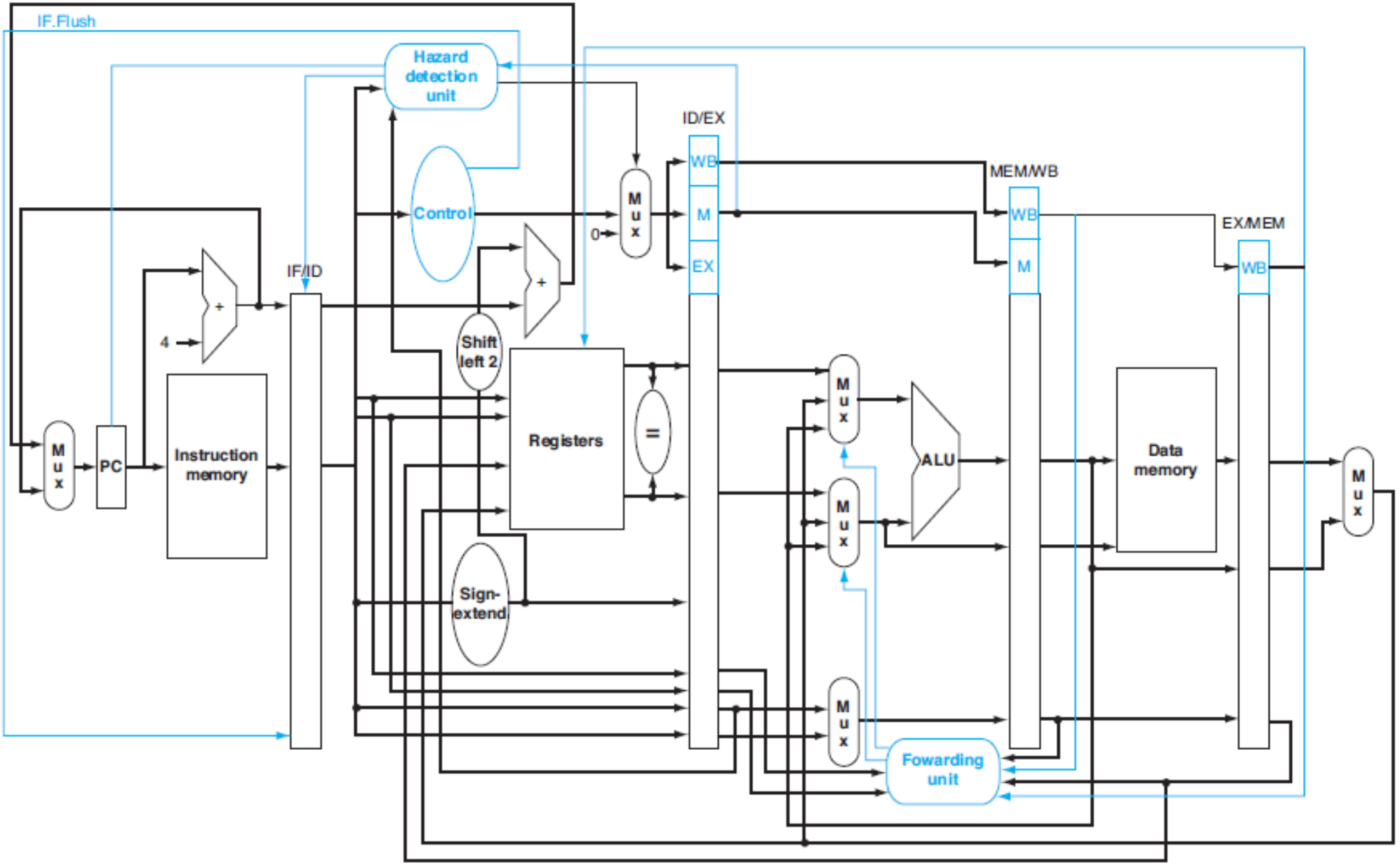
Week 7

Still apparently the toughest question

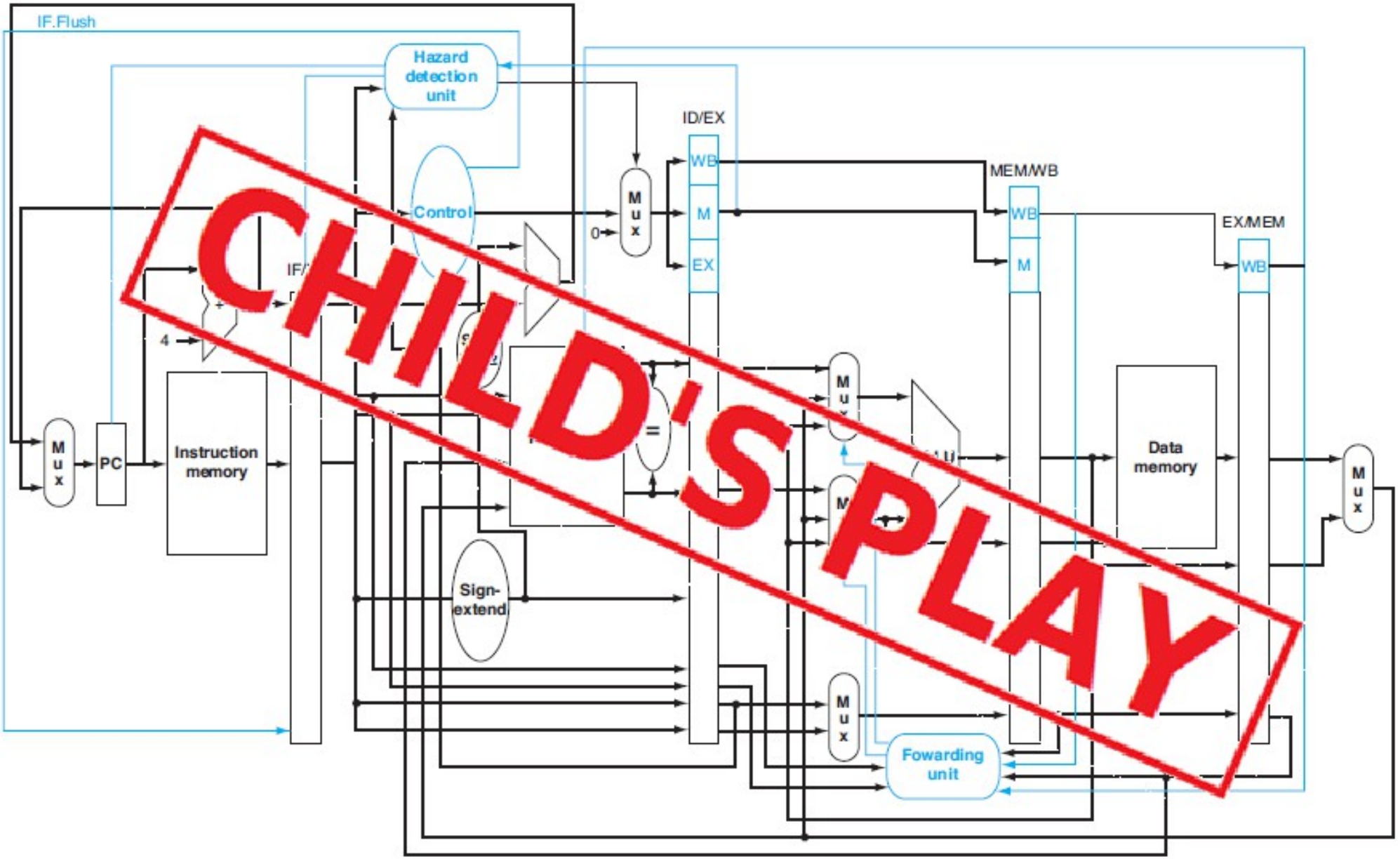
Metric prefixes V·T·E

Prefix		1000^m	10^n	Decimal	English word		Since[n 1]
name	symbol				short scale	long scale	
yotta	Y	1000^8	10^{24}	1 000 000 000 000 000 000 000 000	septillion	quadrillion	1991
zetta	Z	1000^7	10^{21}	1 000 000 000 000 000 000 000	sextillion	thousand trillion	1991
exa	E	1000^6	10^{18}	1 000 000 000 000 000 000	quintillion	trillion	1975
peta	P	1000^5	10^{15}	1 000 000 000 000 000	quadrillion	thousand billion	1975
tera	T	1000^4	10^{12}	1 000 000 000 000	trillion	billion	1960
giga	G	1000^3	10^9	1 000 000 000	billion	thousand million	1960
mega	M	1000^2	10^6	1 000 000	million		1960
kilo	k	1000^1	10^3	1 000	thousand		1795
hecto	h	$1000^{2/3}$	10^2	100	hundred		1795
deca	da	$1000^{1/3}$	10^1	10	ten		1795
		1000^0	10^0	1	one		–
deci	d	$1000^{-1/3}$	10^{-1}	0.1	tenth		1795
centi	c	$1000^{-2/3}$	10^{-2}	0.01	hundredth		1795
milli	m	1000^{-1}	10^{-3}	0.001	thousandth		1795
micro	μ	1000^{-2}	10^{-6}	0.000 001	millionth		1960
nano	n	1000^{-3}	10^{-9}	0.000 000 001	billionth	thousand millionth	1960
pico	p	1000^{-4}	10^{-12}	0.000 000 000 001	trillionth	billionth	1960
femto	f	1000^{-5}	10^{-15}	0.000 000 000 000 001	quadrillionth	thousand billionth	1964
atto	a	1000^{-6}	10^{-18}	0.000 000 000 000 000 001	quintillionth	trillionth	1964
zepto	z	1000^{-7}	10^{-21}	0.000 000 000 000 000 000 001	sextillionth	thousand trillionth	1991
yocto	y	1000^{-8}	10^{-24}	0.000 000 000 000 000 000 000 001	septillionth	quadrillionth	1991

The Good Ole Pipeline



The Good Ole Pipeline



Advanced Pipelining

- How do we improve the performance of the pipeline?
- Start by reevaluating old faithful:
  - ET = IC * CPI * CT
- How do we reduce ET?

Advanced Pipelining

- How do we improve the performance of the pipeline?
- Start by reevaluating old faithful:
  - ET = IC * CPI * CT
- How do we reduce ET?
  - The microarchitecture can only really influence CPI and CT

Advanced Pipelining

- CT?
- The pipelined datapath is preferred over the single cycle datapath because of the significant reduction of CT.
- In general, the CT can be reduced by splitting the datapath into more stages.
- We can decrease the CT by increasing the number of stages AKA having a “deeper pipeline”.

Advanced Pipelining

- CPI?
- Well... the CPI is already 1.
- Mission complete.

Advanced Pipelining

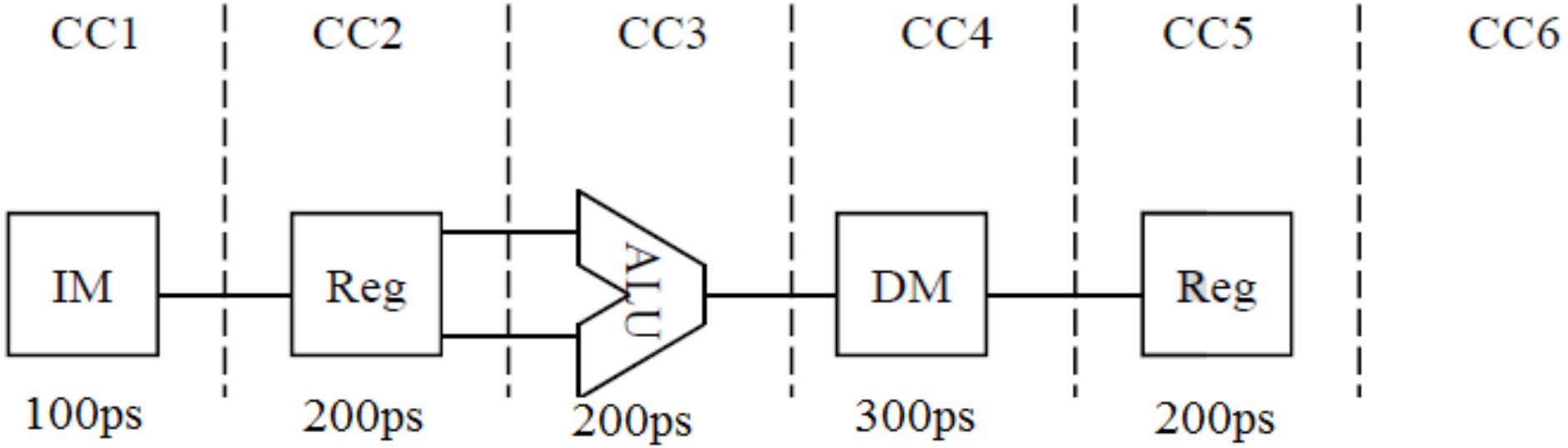
- Well...
- Typically we've been using cycles per instruction. How could an instruction take fewer than 1 cycles?
- Think of the inverse unit of measure, instructions per cycle (IPC, not Interprocess Communication)
- If we could complete more than one instruction in one cycle (for example, if we had two full pipelines), we could complete 2 IPC.
- This would correspond to a CPI of .5!

Advanced Pipelining

- Can improve CPI by modifying the pipeline so that multiple instructions can pass through each stage simultaneously.

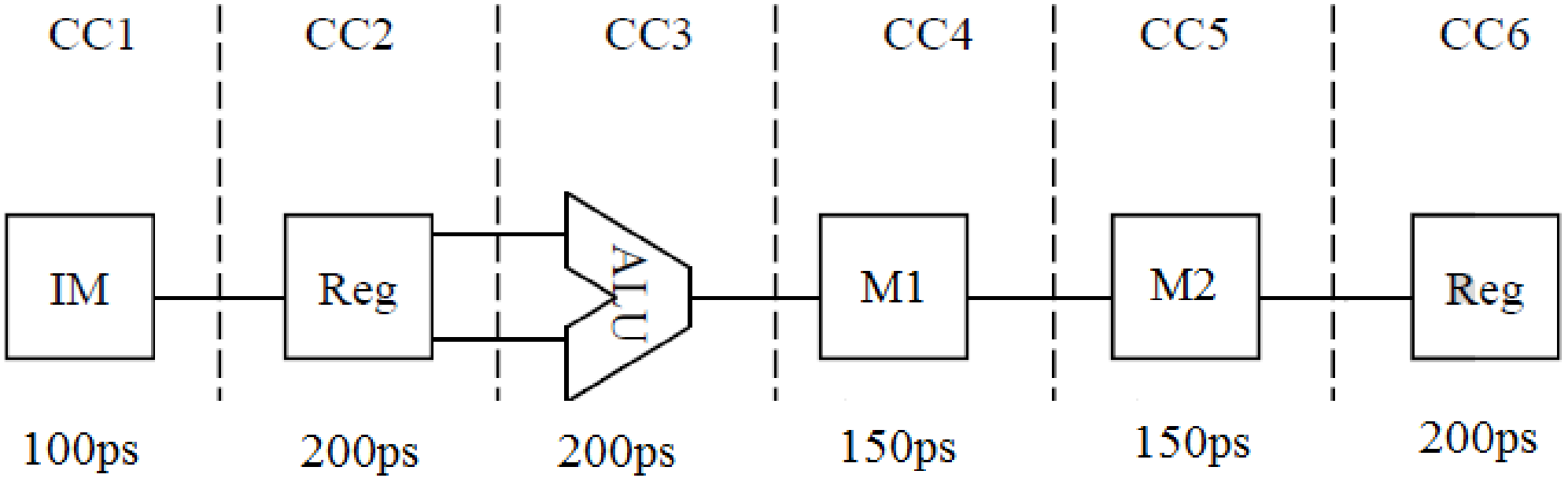
Advanced Pipelining: More Stages

- Also cleverly known as... super pipelining.
- Theory: More pipeline stages means lower clock time.
- The bottleneck here is the MEM stage. Let's split it into two stages.



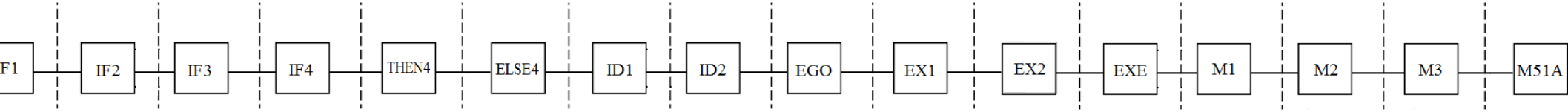
Advanced Pipelining: More Stages

- The 300ps MEM stage is split into two 150ps M1 and M2 stages.
- Now, the memory modules are not bottlenecks.
- The CT is 200ps, a 33% reduction. Not bad.



Advanced Pipelining: More Stages

- Your imagination is the limit!



- ...or could there be some tradeoffs?

Advanced Pipelining: More Stages

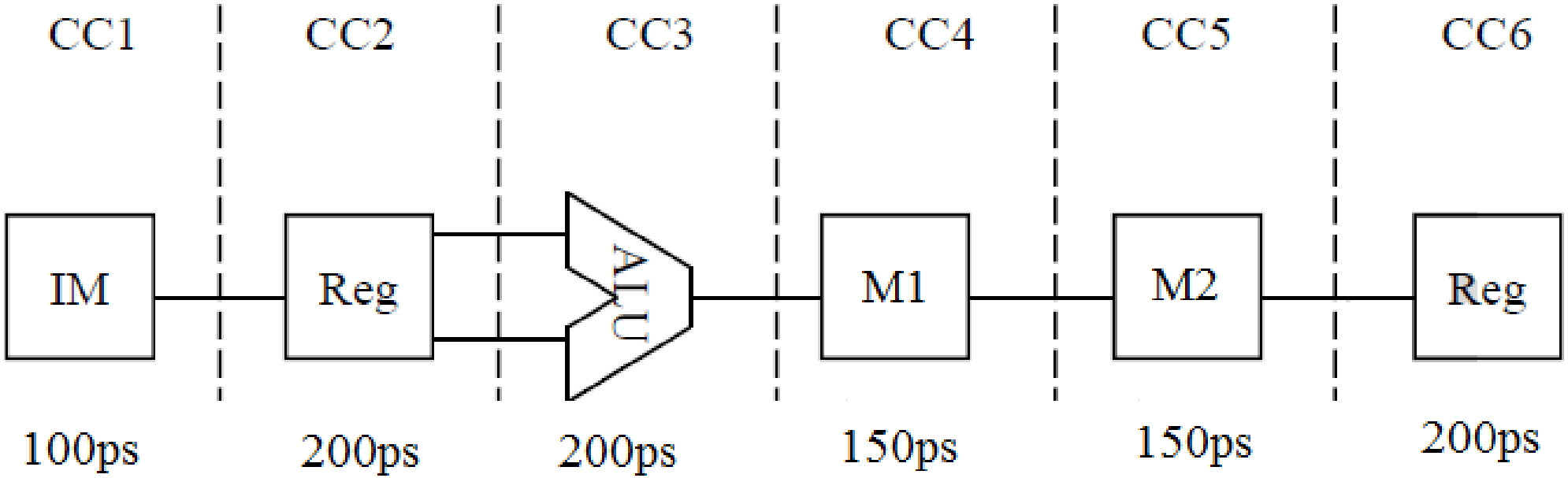
- Tradeoffs?
  - Keep in mind that there are latches in between each stage. Eventually with enough stages, latch latency is going to dominate.
  - Increasing the number of stages is a very easy way to introduce more data and control hazards.
  - It also will become increasingly difficult to find ways to split the stages.
  - CT goes down.
  - CPI remains 1?

Advanced Pipelining: More Stages

- Tradeoffs?
  - Theoretically, the CPI remains 1 in the ideal case.
  - Just like how the CPI in the original pipeline was “theoretically” 1.
  - However, Data and Control Hazards prevented it from truly being 1 even in the simpler pipeline.
  - How about in a more advanced one?

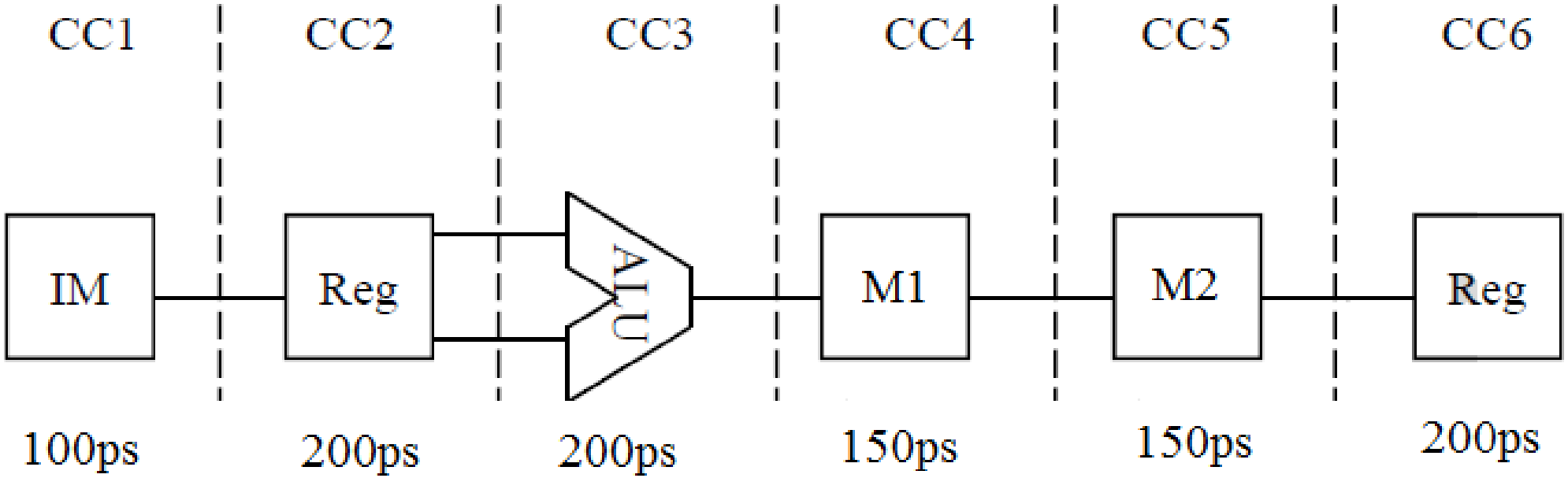
Advanced Pipelining: More Stages

- Assume some similarity to the simple pipeline
  - We can forward from EX/M1, M1/M2, and M2/WB latch to the EX stage.



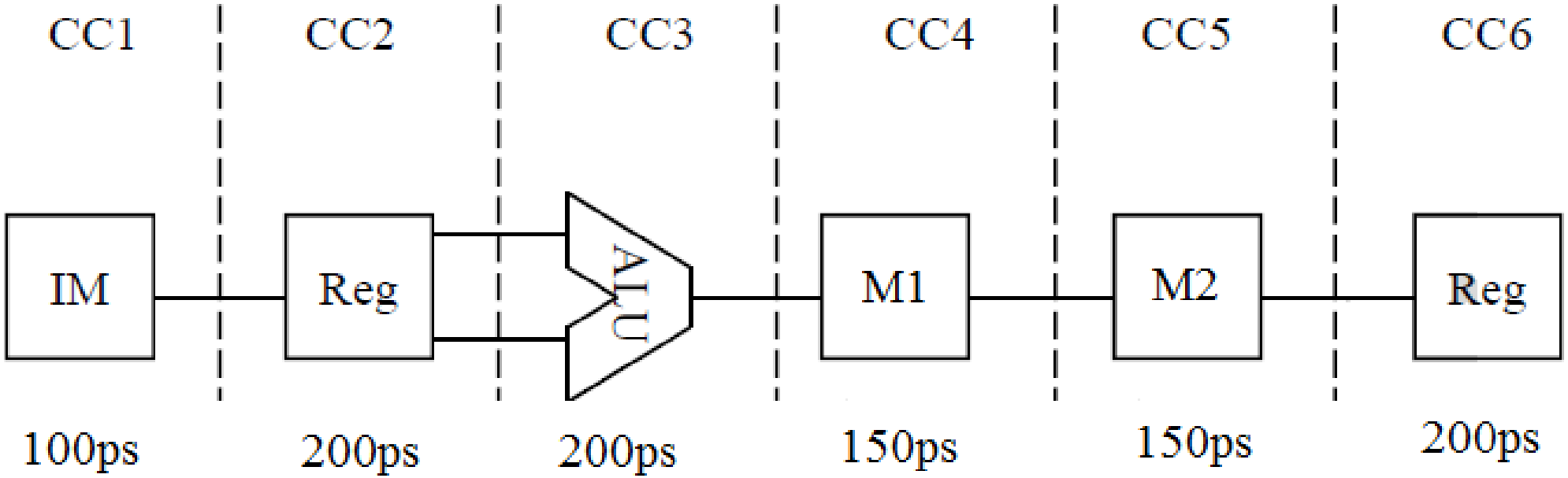
Advanced Pipelining: More Stages

- What effect will this have on load dependencies? What do we need to know before we can answer that?



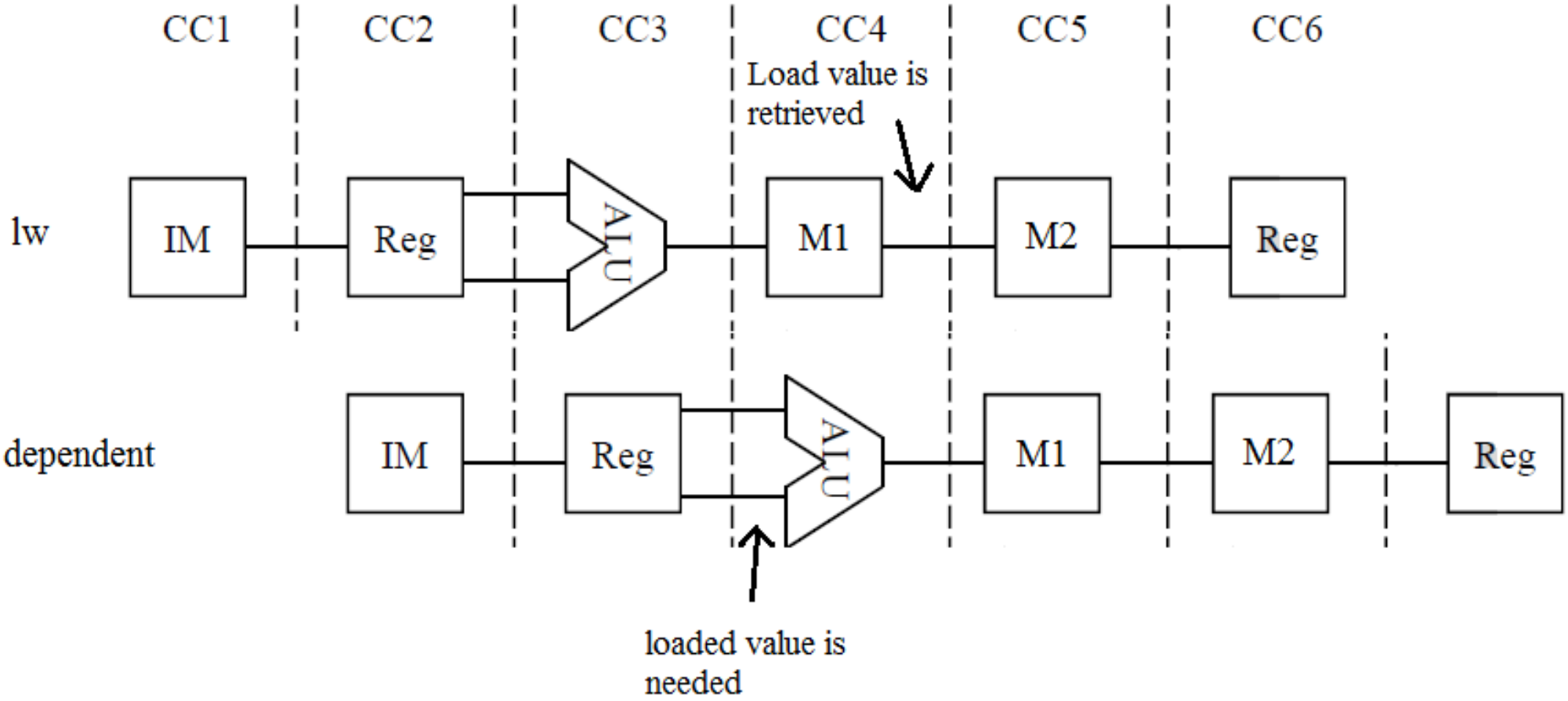
Advanced Pipelining: More Stages

- Let's assume that lw has the result retrieved in the M1 stage.
- Consider a load followed by a dependent instruction.



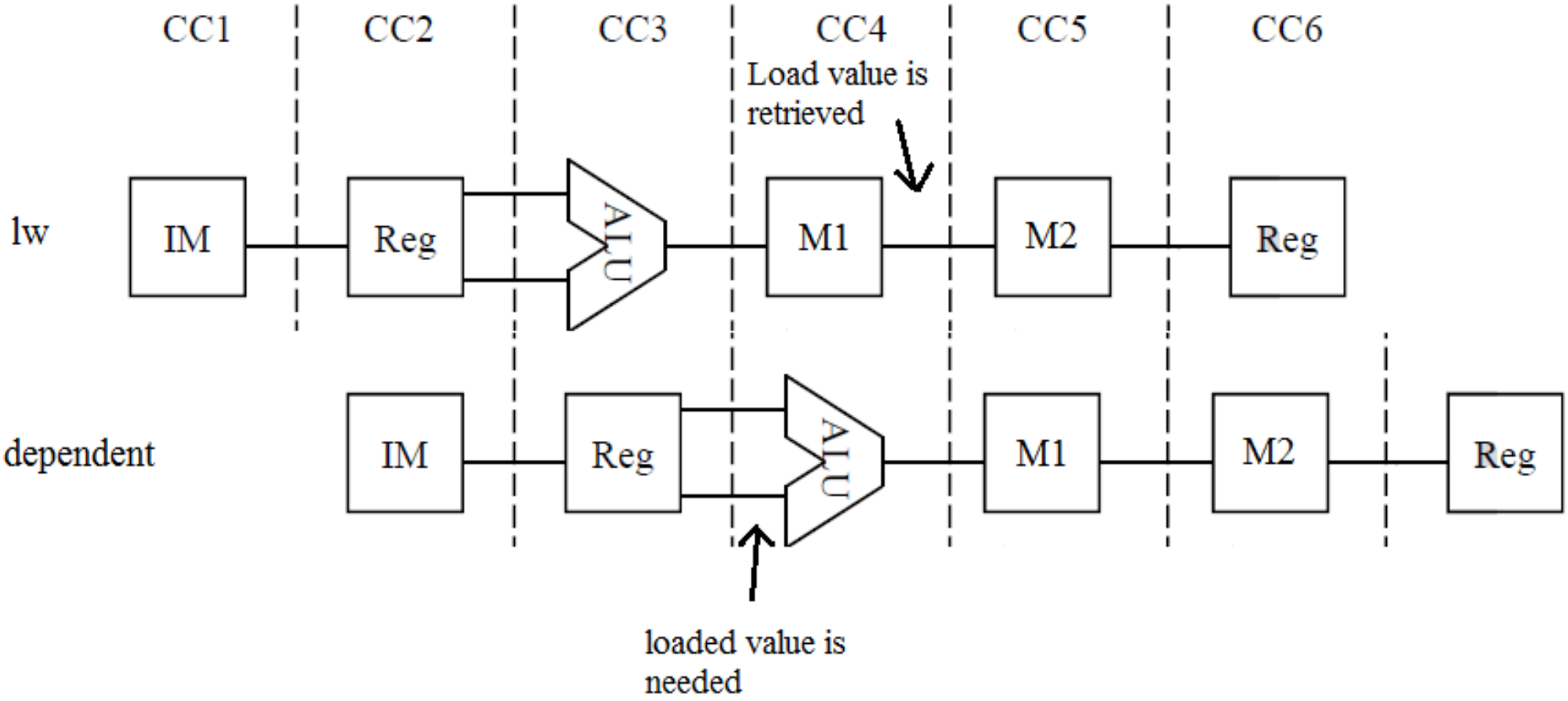
Advanced Pipelining: More Stages

- This is the same as in the original pipeline. 1 stall needed. Not bad.



Advanced Pipelining: More Stages

- What if load is completed in M2?

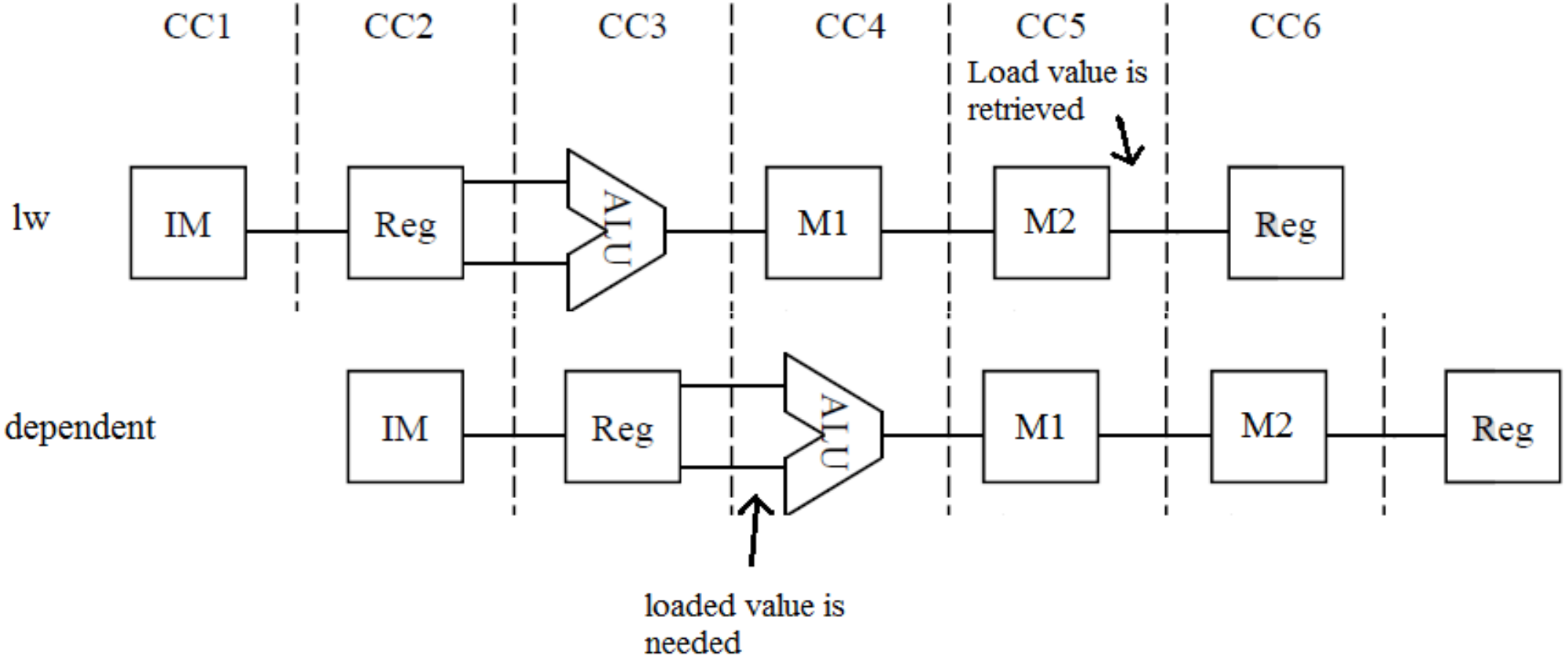


Advanced Pipelining: More Stages

- What if load is completed in M2?

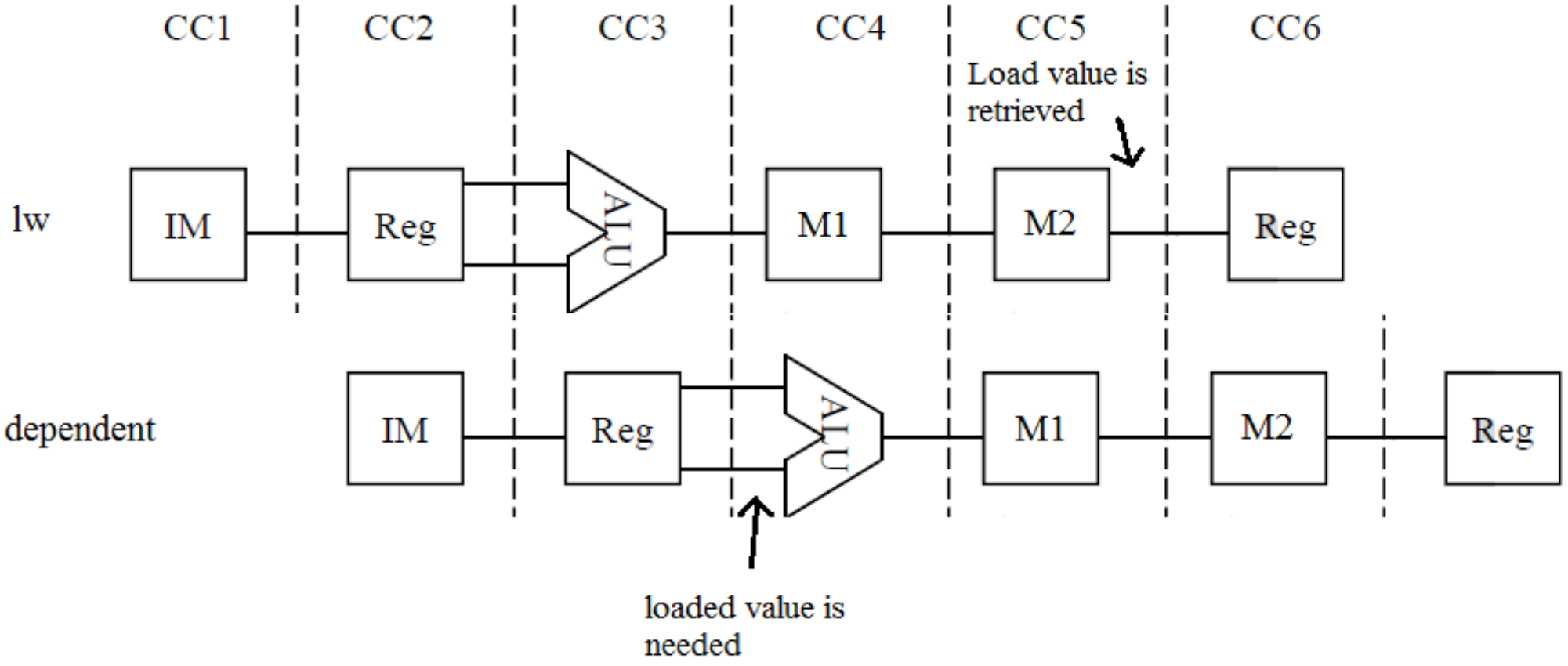
lw $tx IMM($ty)
add $tz $tx, $tx
2 stalls necessary

lw $tx IMM($ty)
independent insn
add $tz $tx, $tx
1 stalls necessary



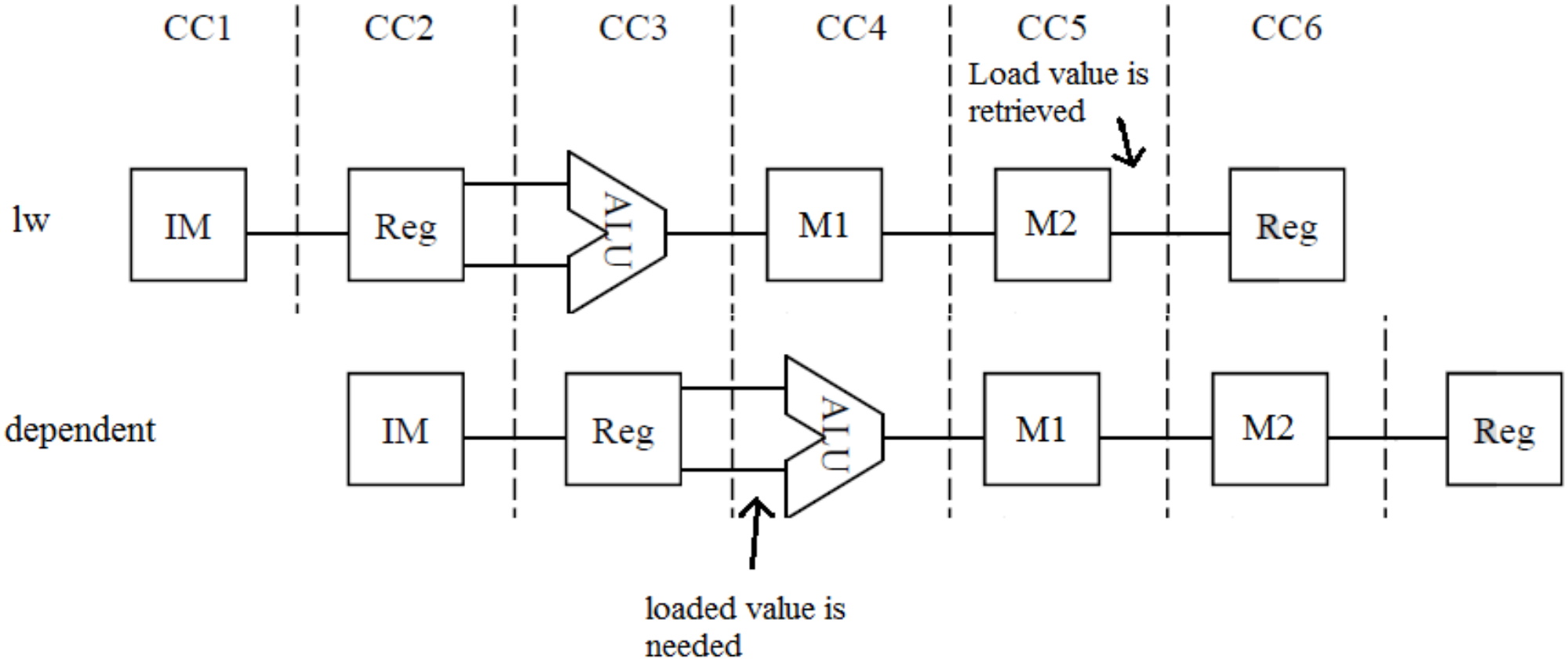
Advanced Pipelining: More Stages

- Are R-Type dependencies affected?
  add $tx, __, ___
  add __, $tx, $tx



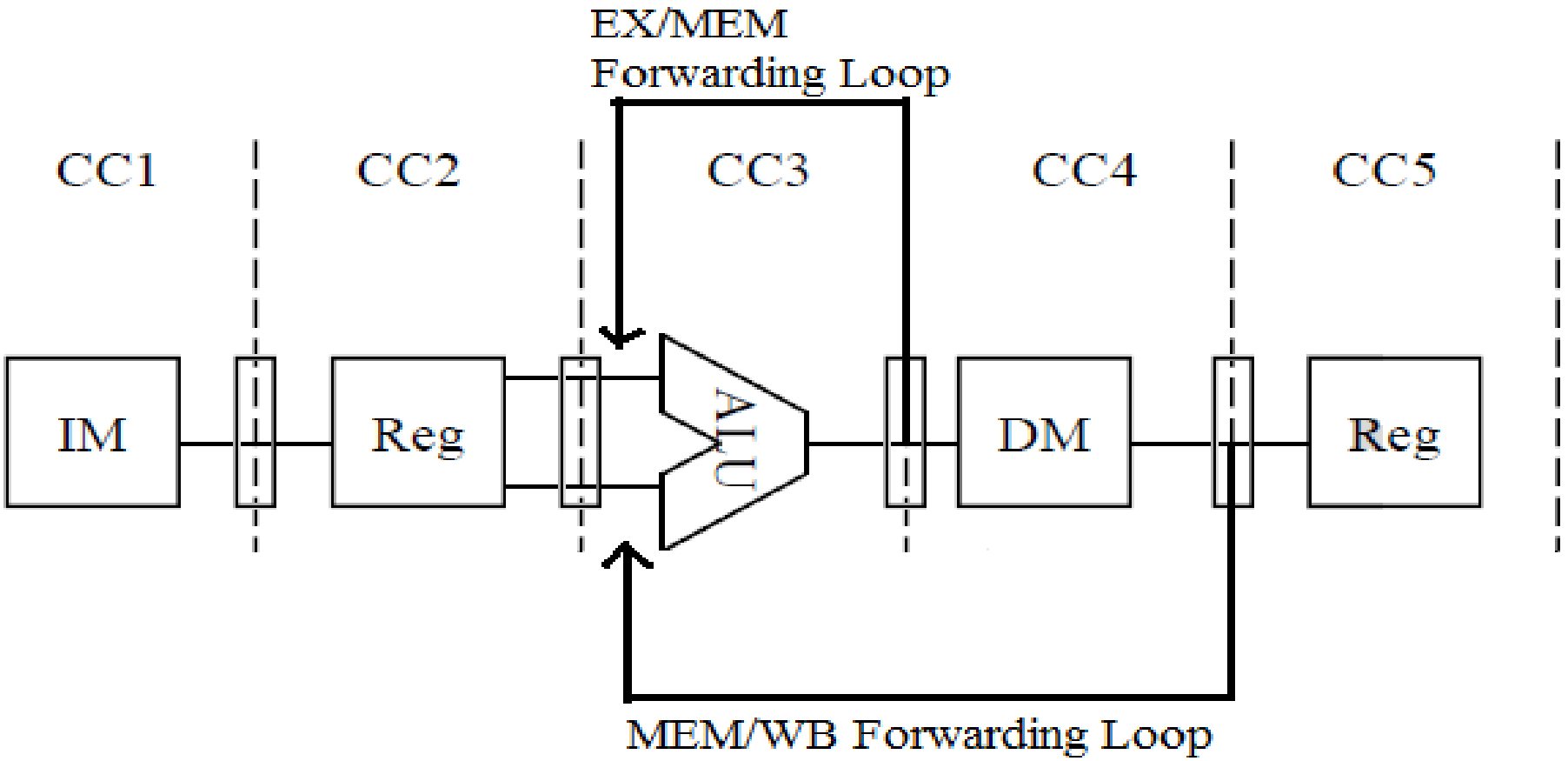
Advanced Pipelining: More Stages

- Are R-Type dependencies affected?
  - No, results of R-Type instructions are still completed in CC3 to be used in CC4.



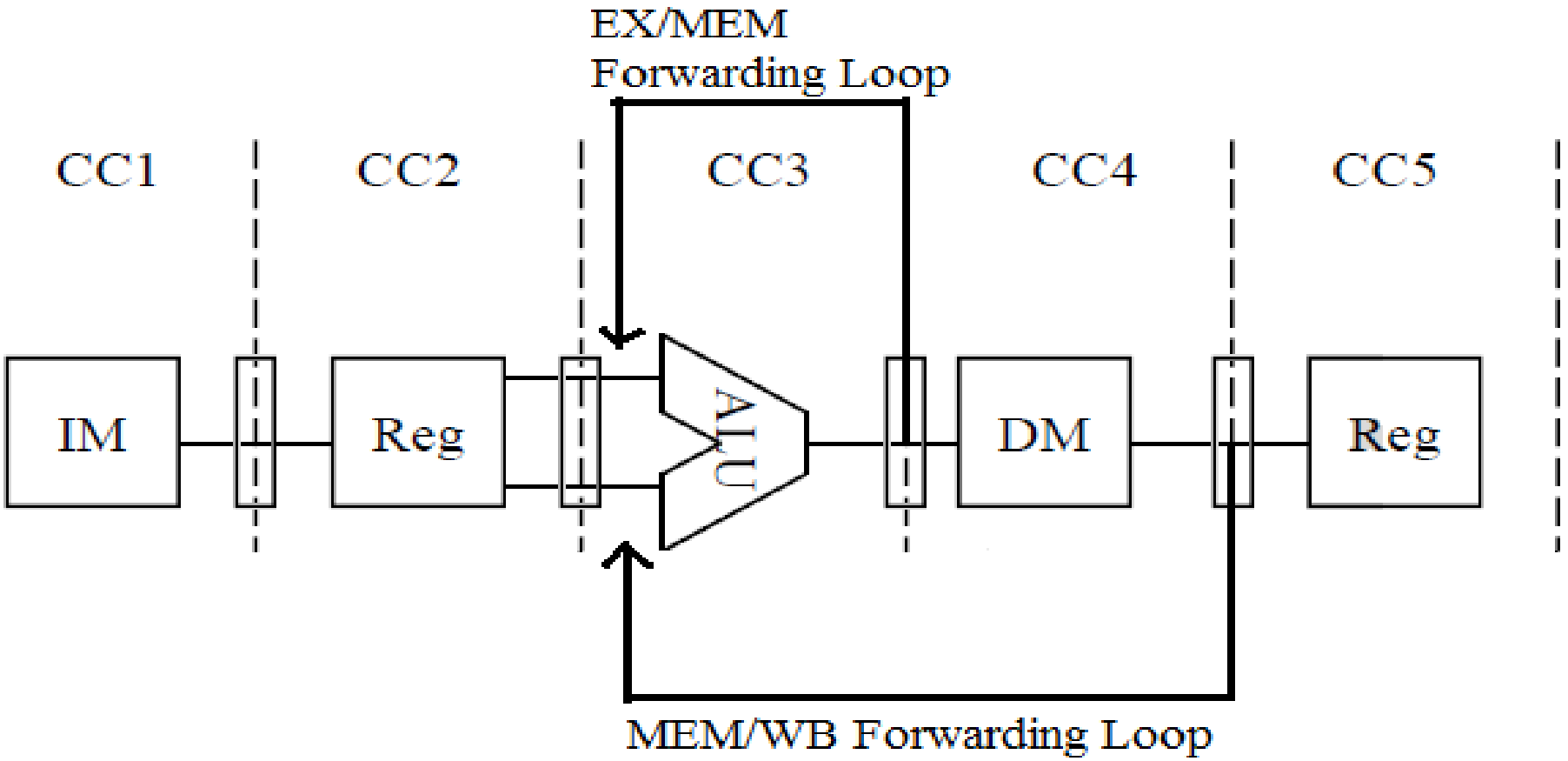
Advanced Pipelining: More Stages

- In general, consider the how the control and data can loop back along the datapath.
- If a stage that is between a loop source and destination is split, it will affect the behavior of the loop.



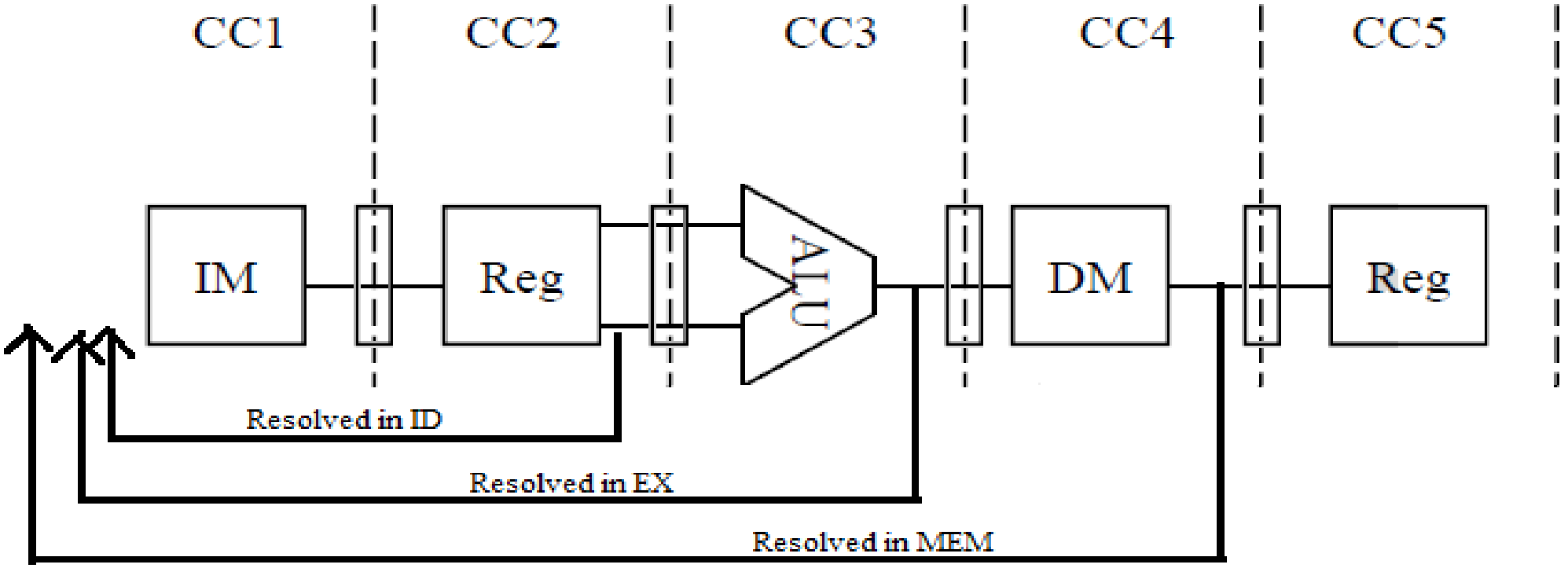
Advanced Pipelining: More Stages

- For example, increasing the number of EX stages will influence both R-Type and load data hazards, but increasing the number of MEM stages will only affect load hazards.



Advanced Pipelining: More Stages

- This is also true for branching.



Advanced Pipelining: Multi-Issue

- Sadly, not called “super issue”.
- With the normal pipeline, the goal was to have multiple instructions in the datapath at once.
- Now, the goal is to have multiple instructions in each stage of the datapath at once.
- This means multiple instructions entering and exiting the pipeline in each cycle.

Advanced Pipelining: Multi-Issue

- Note: Of course, multiple instructions executed simultaneously can be done with multiple cores or processors but for now, we're considering just a single superscalar processor.

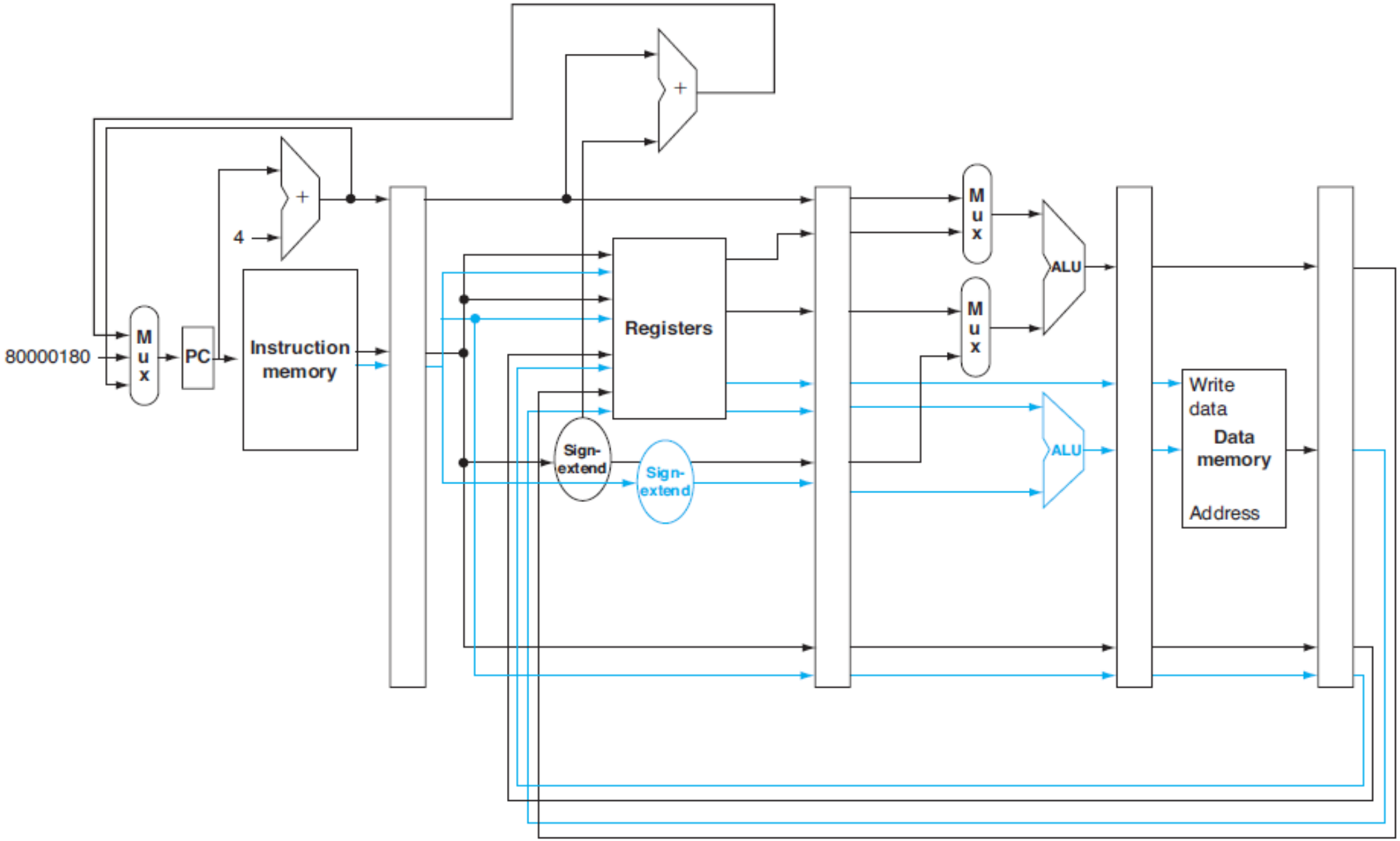
Advanced Pipelining: Multi-Issue

- Code is presented as a single sequence of instructions. How do we determine which instructions can be executed simultaneously?
- Static Multi-Issue
- Dynamic Multi-Issue

Advanced Pipelining: Static Multi-Issue

- Static Multi-Issue
  - Compiler examines the code and determines the set of instructions that can be run simultaneously (issue packet)
  - Compiler handles data and control hazards, even applying static branch predication to rearrange instructions, in this case, into pairs.
  - Requires a specialized datapath
  - The datapath will restrict on what types of instructions can be executed simultaneously.

Advanced Pipelining: Static Multi-Issue



Advanced Pipelining: Static Multi-Issue

- Blue path corresponds to memory path while black path corresponds to R-Type/ALU path.
- Register file can now read and write twice as many values.
- Blue path uses an adder ALU in EX stage to do immediate offset.
- Black path uses full ALU in EX.
- Branches seem to be resolved in ID
- Full forwarding

Advanced Pipelining: Static Multi-Issue

- General restriction:
  - In each issue packet, you can have one R-Type/branch (add, sub, beq, etc.) instruction and one memory operation (lw, sw).
  - Because these will run through the stages of the pipeline simultaneously, the instructions within a single issue packet must be free of read after write true dependencies.

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Data Dependencies?

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Which instructions have their ordering enforced?

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Which instructions have their ordering enforced? Instruction 1, 2, and 3, must be done in a strict order because they have true dependencies. Additionally, because of load dependencies, they must be separated by an instruction.

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. ___ | lw $t0, 8($s0)
2. ___ | ___
3. ___ | lw $t0, 0($t0)
4. ___ | ___
5. add $s2, $t0, $s2 | ___
6. ___ | ___

What about the addi and the bne? Does their position or ordering matter?

Advanced Pipelining: Static Multi-Issue

Consider:

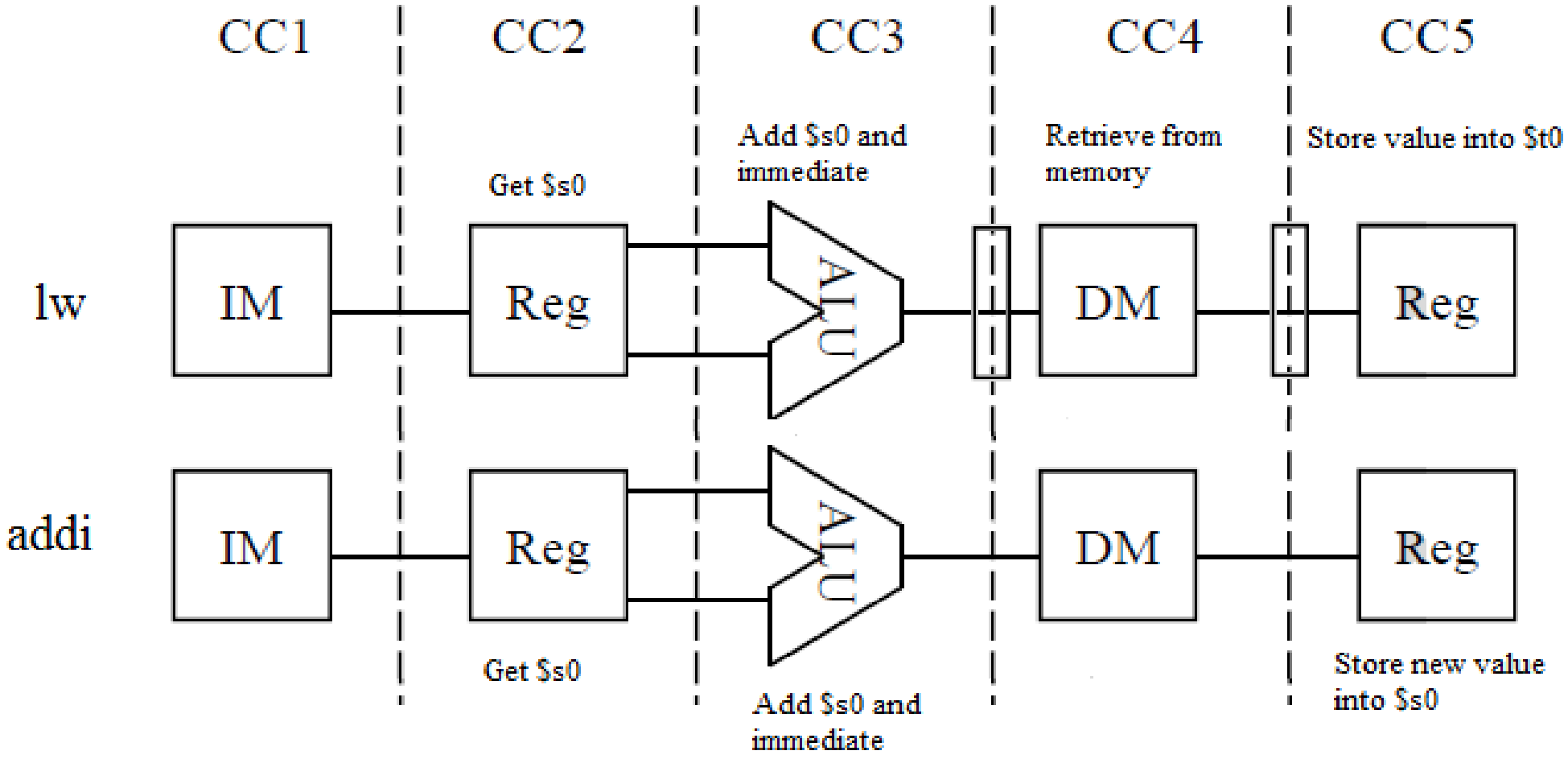
LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. ___ | lw $t0, 8($s0)
2. ___ | ___
3. ___ | lw $t0, 0($t0)
4. ___ | ___
5. add $s2, $t0, $s2 | ___
6. ___ | ___

The addi uses only $s0 so it is definitely independent from the second lw and the add. However, it can also be placed in the same slot as the first lw because they will both read from the register file in the same cycle

Advanced Pipelining: Static Multi-Issue



Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. ____ | lw $t0, 8($s0)
2. ____ | ____
3. ____ | lw $t0, 0($t0)
4. addi $s0, $s0, 16 | ____
5. add $s2, $t0, $s2| ____
6. bne $s0, $s1, LABEL | ____

Unfortunately, regardless of where we put the addi, it doesn't help much. The bne must come after the addi and at the end of the instructions.

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. nop | lw $t0, 8($s0)
2. nop | nop
3. nop | lw $t0, 0($t0)
4. addi $s0, $s0, 16 | nop
5. add $s2, $t0, $s2 | nop
6. bne $s0, $s1, LABEL | nop

- The original snippet of code would have taken 7 cycles (two stalls for the two load dependencies).
- We've reduced that to... 6 cycles.
- Stop the presses...

Advanced Pipelining: Static Multi-Issue

- When considering static multi-issue, we have to consider additional compiler optimizations to make it useful.
- In particular one useful one is loop unrolling.
- At a high level this essentially means to do fewer loops, but more operations per loop.
- Consider the following, where the variable size is 100.

Advanced Pipelining: Static Multi-Issue

Original:

```
for(i = 0; i < size; i++)
{
  a[i] = i;
}
```

This loop iterates 100 times, doing one instruction each iteration.

Loop unrolling after:

```
for(i = 0; i < size; i+=2)
{
  a[i] = i;
  a[i+1] = i + 1;
}
```

This loop iterates 50 times, doing two instructions each iteration.

Advanced Pipelining: Static Multi-Issue

Original:

```
for(i = 0; i < size; i++)
{
  a[i] = i;
}
```

This loop must do the loop overhead (compare i with size, i++) 100 times.

Loop unrolling after:

```
for(i = 0; i < size; i+=2)
{
  a[i] = i;
  a[i+1] = i + 1;
}
```

This loop must do the loop overhead (compare i with size, i+=2) 50 times.

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Unrolled:
LABEL
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
addi $s0, $s0, 16
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
addi $s0, $s0, 16
bne $s0, $s1, LABEL

- Let's unroll the loop. Now each loop iteration will actually do the original loop body (lw, lw, add, addi) twice.
- Note: This only works if we assume that the loop will be taken a multiple of 2 times.

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Unrolled:
LABEL
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
addi $s0, $s0, 16
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
addi $s0, $s0, 16
bne $s0, $s1, LABEL

- Part of the point of this was to reduce the amount of overhead. Instead of doing addi twice, we can do the addi only once.

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Unrolled:
LABEL
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
addi $s0, $s0, 32
bne $s0, $s1, LABEL

- Because now s0 is only being changed once for every two operations, we must adjust the unrolled loop accordingly. For example, as written, these two lw would access the same value, which is not the original intent.

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Unrolled:
LABEL
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
lw $t0, 24($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
addi $s0, $s0, 32
bne $s0, $s1, LABEL

- The second lw must add 16 to the offset (which is the value that s0 would have been changed by with the addi).

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Unrolled:
LABEL
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
lw $t0, 24($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
addi $s0, $s0, 32
bne $s0, $s1, LABEL

- Another thing to notice is the fact that we use t0 in the first three instructions and then reassign a value to t0, which creates dependencies between instructions from the two loop iterations. How do we resolve this?

Advanced Pipelining: Static Multi-Issue

Consider:

LABEL:
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. addi $s0, $s0, 16
5. bne $s0, $s1, LABEL

Unrolled:
LABEL
lw $t0, 8($s0)
lw $t0, 0($t0)
add $s2, $t0, $s2
lw $t1, 24($s0)
lw $t1, 0($t1)
add $s2, $t1, $s2
addi $s0, $s0, 32
bne $s0, $s1, LABEL

- Rename the registers from the second loop (from t0 to t1). Now there is more parallelism we can exploit.

Advanced Pipelining: Static Multi-Issue

- Note: What if we do this loop an odd number of times?
- We would need some additional overhead of doing the remaining instruction.
- This is a general cost of loop unrolling but let's ignore that.

Advanced Pipelining: Static Multi-Issue

Unrolled:
LABEL
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. lw $t1, 24($s0)
5. lw $t1, 0($t1)
6. add $s2, $t1, $s2
7. addi $s0, $s0, 32
8. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. ___ , ___
2. ___ , ___
3. ___ , ___
4. ___ , ___
5. ___ , ___
6. ___ , ___
7. ___ , ___
8. ___ , ___
9. ___ , ___

- What orderings must be enforced?

Advanced Pipelining: Static Multi-Issue

Unrolled:
LABEL
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. lw $t1, 24($s0)
5. lw $t1, 0($t1)
6. add $s2, $t1, $s2
7. addi $s0, $s0, 32
8. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. ___ , ___
2. ___ , ___
3. ___ , ___
4. ___ , ___
5. ___ , ___
6. ___ , ___
7. ___ , ___
8. ___ , ___
9. ___ , ___

- What orderings must be enforced? 1 → gap → 2 → gap → 3 and 4 → gap → 5 → gap → 6. However, these operations can be interleaved now.

Advanced Pipelining: Static Multi-Issue

Unrolled:
LABEL
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. lw $t1, 24($s0)
5. lw $t1, 0($t1)
6. add $s2, $t1, $s2
7. addi $s0, $s0, 32
8. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. ___ , lw $t0, 8($s0)
2. ___ , lw $t1, 24($s0)
3. ___ , lw $t0, 0($t0)
4. ___ , lw $t1, 0($t1)
5. add $s2, $t0, $s2, ___
6. add $s2, $t1, $s2, ___
7. ___ , ___
8. ___ , ___
9. ___ , ___

- What about the addi and bne?

Advanced Pipelining: Static Multi-Issue

Unrolled:
LABEL
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. lw $t1, 24($s0)
5. lw $t1, 0($t1)
6. add $s2, $t1, $s2
7. addi $s0, $s0, 32
8. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. nop | lw $t0, 8($s0)
2. nop | lw $t1, 24($s0)
3. nop | lw $t0, 0($t0)
4. addi $s0, $s0, 32 | lw $t1, 0($t1)
5. add $s2, $t0, $s2 | nop
6. add $s2, $t1, $s2 | nop
7. bne $s0, $s1, LABEL | nop

- What about the addi and bne?

Advanced Pipelining: Static Multi-Issue

Unrolled:
LABEL
1. lw $t0, 8($s0)
2. lw $t0, 0($t0)
3. add $s2, $t0, $s2
4. lw $t1, 24($s0)
5. lw $t1, 0($t1)
6. add $s2, $t1, $s2
7. addi $s0, $s0, 32
8. bne $s0, $s1, LABEL

ALU | Memory

LABEL:
1. nop | lw $t0, 8($s0)
2. nop | lw $t1, 24($s0)
3. nop | lw $t0, 0($t0)
4. addi $s0, $s0, 32 | lw $t1, 0($t1)
5. add $s2, $t0, $s2 | nop
6. add $s2, $t1, $s2 | nop
7. bne $s0, $s1, LABEL | nop

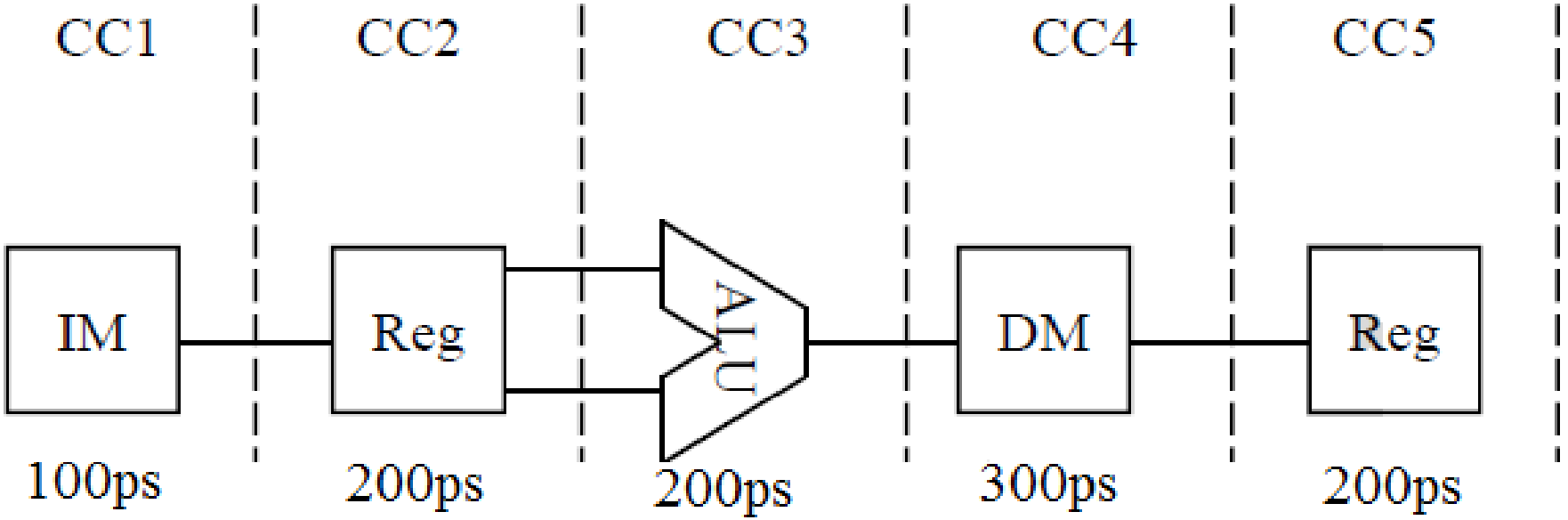
- Originally, a single loop body took 6 cycles (not counting the bne). Now TWO iterations of the loop takes 6 cycles (not counting the bne).
- Stop the presses!

Clever Transition

- So far we have considered:
  - ISA
  - Microarchitecture
    - ALU
    - Single Cycle Datapath
    - Pipelined Datapath
- But we've sort of neglected memory.

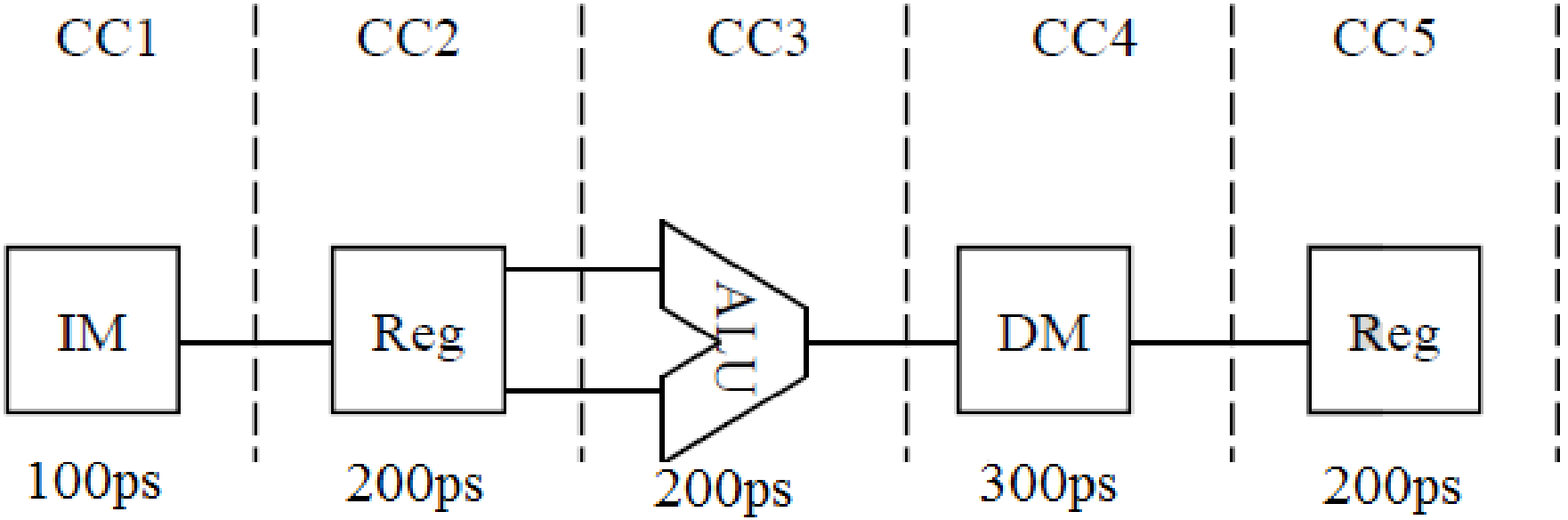
Memory

- When we consider pipeline stages, we see something like the following.
- The DM or MEM stage is the stage in which you access memory.
- However, recall the memory hierarchy.



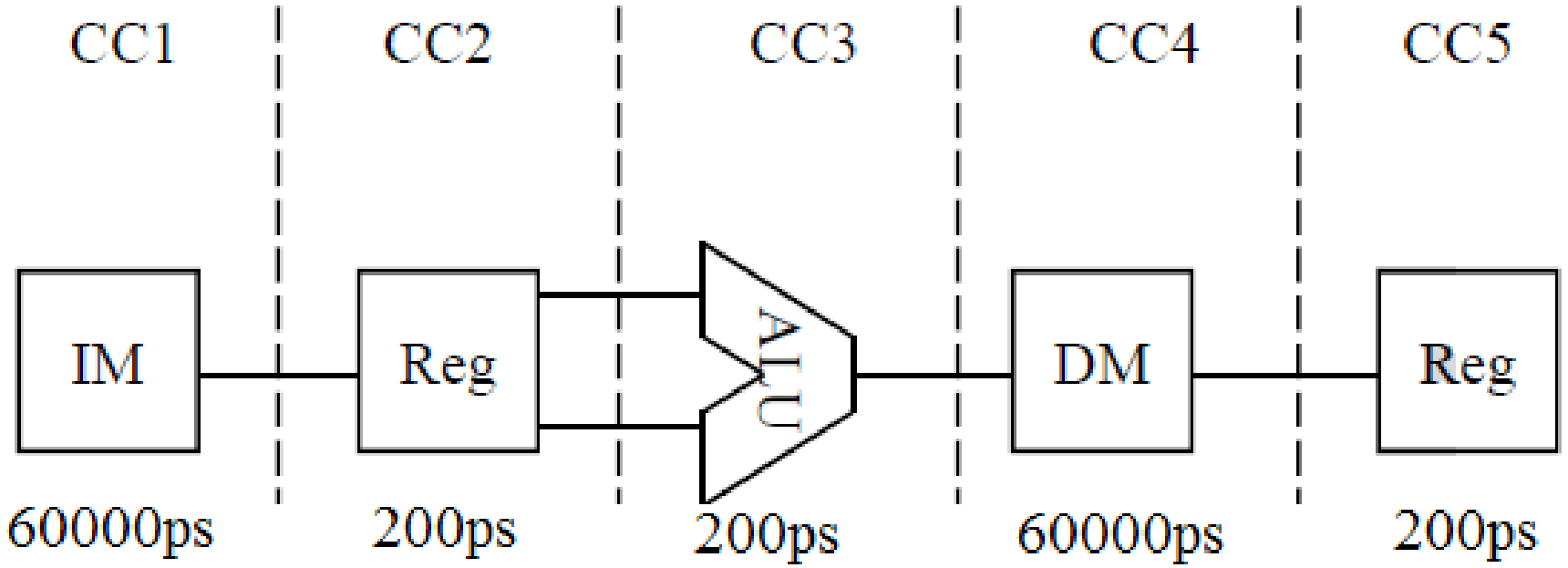
Memory

- The Memory Hierarchy: It's the triangle. You know the one.
- The top of the pyramid is registers while RAM is several tiers lower.
- This means, a more accurate timing scheme would be...



Memory

- The Memory Hierarchy: It's the pyramid picture. You know the one.
- The top of the pyramid is registers while RAM is several tiers lower.
- This means, a more accurate timing scheme would be...



Memory

- As you can imagine, it would be pretty unacceptable to have the clock cycle be 60 ns while most of the stages are several orders of magnitude faster than that (those poor common cases!).
- To have a viable pipeline, we need a faster way to access memory.
- Easier said than done (memory wall).

Cache

- Despite being random access, the sheer size and the resulting design of DRAM is preventing it from being fast.
- Instead we would like to be able to access memory with speeds comparable to the other pipeline stages.
- Cache: A small(er) storage unit that holds a subset of memory that is much faster than accessing main memory.

Cache

- The cache can only contain a small set of memory at once? How do we decide what is worthy of going into the cache?
- It would be difficult to attempt to do a sort of global prioritization where you statically select the most “important” or commonly used data.
- Probably going to have to populate cache dynamically. If so, what guiding principle do we need to use?

Cache

- Spatial Locality:
  - If you access some data in some memory address, you are likely to access the data in nearby memory addresses.
  - Ex. Array accesses, sequential instructions
- Temporal Locality:
  - If you access some data, you are likely to use that data again soon.
  - Ex. Objects, instructions in a loop

Cache

- When we want to access memory, we first go through the cache.
- If the memory address is in the cache, then great.
- If loading and the memory address isn't in the cache, then copy the “block” that this address corresponds to into the cache, then read.
- If storing... well, more on that later.

Cache

- Definitions
- Cache blocks:
  - A chunk of bytes in memory that are adjacent.
  - Ex: If block size of 2^5 bytes, then each block is 32 bytes long.
  - Block 0: M[0] to M[31]
  - Block 1: M[32] to M[63]
  - Etc...
  - Note: M[15] belongs to block 0.

Cache

- Definitions
- Cache blocks:
  - A chunk of bytes in memory that are adjacent.
  - Ex: If block size of 2^5 bytes, then each block is 32 bytes long.
  - If the address space is 2^{32} and blocks sizes are 2^5, then there are 2^{27} blocks in memory.
  - All operations performed on cache (eviction, moving), are done on the granularity of blocks.

Cache

- Definitions
- Each cache consists of an “array” of “sets”.
- Each set consists of one or more cache blocks.
- Each block has a corresponding tag, valid, and dirty (sometimes) bit.
- A tag is like an id that can be used to identify a cache block.

Direct Mapped Cache

Set/Index	Valid	Dirty	Tag	Data
0				
1				
2				
3				
...				
2^n - 1				

2^n sets

1 data block, AKA 1-way/direct mapped

- How do we decide where each block from memory should go?
- We determine a mapping from the physical address

Direct Mapped Cache

- How do we decide where each block from memory should go?
- We determine a mapping from the physical address.
- Each address is split into three components: Tag, Index, Offset
- For example, the address:
- ...11010110101110
- might be split into:
- ...11010 11010 1110
- Tag Index Offset

Direct Mapped Cache

- Tag: The “id” for the block
- Index: Which set the block belongs to
- Offset: Which byte within the block that is being accessed.

Direct Mapped Cache

Set/Index	Valid	Dirty	Tag	Data
000				
001				
010				
011				
100				
101				
110				
111				

- As an example, let's consider a Direct Mapped Cache with 8 (2^3) sets, and 8 (2^3) byte blocks.
- Consider the case where the addressable space is 2^8

Direct Mapped Cache

- As an example, let's consider a Direct Mapped Cache with 8 (2^3) sets, and 8 (2^3) byte blocks.
- Consider the case where the addressable space is 2^8.
- How do we translate an address:
- 0000 0000
- ...into a cache location?

Direct Mapped Cache

- 8 (2^3) sets, 8 (2^3) byte blocks, 2^8 addressable space.
- If each block is 8 bytes, we need to be able to access each byte individually. This means we need 3 offset bits.
- There are 8 sets. We also need 3 bits to index into the 8 different sets.
- This leaves 2 bits to be the tag.
- 00 000 000
- T I O

Direct Mapped Cache

- Say we had:
- int x[16];
- where x begins at address 0. Then we accessed each element in increasing order:
- x[0]
- x[1]
- x[2]
- …
- What would this look like in the cache?

Direct Mapped Cache

- Say we had:
- int x[16];
- where x begins at address 0. Then we accessed each element in increasing order:
- x[0] = 00 000 000
- x[1] = 00 000 100
- x[2] = 00 001 000
- …
- What would this look like in the cache?

Direct Mapped Cache

- Access order:
- 00 000 000
- 00 000 100
- 00 001 000
- 00 001 100
- …
- 00 111 100

Set/Index	Valid	Dirty	Tag	Data
000	0			
001	0			
010	0			
011	0			
100	0			
101	0			
110	0			
111	0			

Direct Mapped Cache

- Access order:
- 00 000 000
- 00 000 100
- 00 001 000
- 00 001 100
- …
- 00 111 100

Set/Index	Valid	Dirty	Tag	Data
000	1		00	x[0], x[1]
001	1		00	x[2], x[3]
010	1		00	x[4], x[5]
011	1		00	x[6], x[7]
100	1		00	x[8], x[9]
101	1		00	x[10], x[11]
110	1		00	x[12], x[13]
111	1		00	x[14], x[15]

Direct Mapped Cache

- For each access, you bring in 8 bytes. This means that in the int array access, for each miss, two elements in the array are brought into the cache. This means a hit rate of 50%. Not bad.
-

Direct Mapped Cache

- Now consider:
- int x[16];
- int y[16];
- Where x begins at address 0 and y begins at address 64.

```
for(int i = 0; i < 16; i++)
{
  y[i] = x[i];
}
```

What is the pattern of access now?

Direct Mapped Cache

- What is the pattern of access now?
- X[0] = 00 000 000
- Y[0] = 01 000 000
- X[1] = 00 000 100
- Y[1] = 01 000 100

  X[2] = 00 001 000
- Y[2] = 01 001 000
- X[3] = 00 001 100
- Y[3] = 01 001 100
- …

Direct Mapped Cache

- What is the pattern of access now?
- X[0] = 00 000 000
- Y[0] = 01 000 000
- X[1] = 00 000 100
- Y[1] = 01 000 100
- …
- Notice that x[i] and y[i] belong to the same set for all i. This means each for each access, the cache is being overwritten. The miss rate is 100%!
- This is thrashing (when alternating accesses happen to map to the same set).

Direct Mapped Cache

- Consider that the cache size is 2^3 sets * 2^3 bytes/set or 64 bytes total.
- x and y are each 64 bytes. We can't hold the entire arrays in the cache, but we should have a better hit rate than 0.
- Enter...

Set Associative Cache

- What if each set could hold two (or more blocks)? Could we resolve this particular thrashing issue?
- Say we want to repurpose our cache which has a data size of 64 bytes.
- Block size: 8 bytes.
- Now two blocks per set.
- How many sets can we have?

Set Associative Cache

- Cache Data Size = No. Sets * Associativity(ways) * Block size
- Associativity is of number of blocks per set.

  64 Bytes = S Sets * 2 Blocks/Set * 8 Bytes/Block
- 64 Bytes = S * 16 Bytes
- S = 4
- Now we have to adjust how we interpret addresses:
- 000 00 000
- T I O

2-Way Associative Cache

Set/Index	Valid	Dirty	Tag	Data	Valid	Dirty	Tag	Data
000	0				0			
001	0				0			
010	0				0			
011	0				0			

2-Way Associative Cache

- Consider the access pattern again:
- X[0] = 000 00 000
- Y[0] = 010 00 000
- X[1] = 000 00 100
- Y[1] = 010 00 100

  X[2] = 000 01 000
- Y[2] = 010 01 000
- X[3] = 000 01 100
- Y[3] = 010 01 100

  After 8 iterations...

-
-
- X[4] = 000 10 000
- Y[4] = 010 10 000
- X[5] = 000 10 100
- Y[5] = 010 10 100

  X[6] = 000 11 000
- Y[6] = 010 11 000
- X[7] = 000 11 100
- Y[7] = 010 11 100

2-Way Associative Cache

Set/Index	Valid	Dirty	Tag	Data	Valid	Dirty	Tag	Data
00	1		000	x[0], x[1]	1		010	y[0], y[1]
01	1		000	x[2], x[3]	1		010	y[2], y[3]
10	1		000	x[4], x[5]	1		010	y[4], y[5]
11	1		000	x[6], x[7]	1		010	y[6], y[7]

- 50% hit rate
- The cache is full, but we're only halfway done.
- The next access is x[8] : 001 00 000 and will map to set 0 again.
- How do we choose which to evict?

Fully Associative Cache

- How do we choose which to evict?
- But before we address that thrilling issue...
- Fully associate cache: only 1 set. The associativity determines how many blocks can fit into the cache.
- If fully associative, address is split into two components, tag and offset.
- Notice that by knowing how the address is split, we can determine block size and the number of sets, but we don't know anything about the associativity.

Fully Associative Cache

- Assuming the same 64 byte cache size and 8 byte cache block, what's the associativity if we convert it to a fully associative cache?

Fully Associative Cache

- Cache Size = No. Sets * Associativity * Block Size
- 64 = 1 * Associativity * 8
- Associativity = 8.

Associativity Tradeoffs

- What are the benefits of using a Direct Mapped Cache?

Associativity Tradeoffs

- What are the benefits of using a Direct Mapped Cache?
  - Easy to know exactly where a block should belong and what block should be evicted
- What's the downside?

Associativity Tradeoffs

- What are the benefits of using a Direct Mapped Cache?
  - Easy to know exactly where a block should belong and what block should be evicted
- What's the downside?
  - Thrashing and the consequent poor performance when a cache is not used well

Associativity Tradeoffs

- What's the benefit of having increased associativity?

Associativity Tradeoffs

- What's the benefit of having increased associativity?
  - The problem of thrashing is reduced. Potentially better use of the cache
- Downside?

Associativity Tradeoffs

- What's the benefit of having increased associativity?
  - The problem of thrashing is reduced. Potentially better use of the cache
- Downside?
  - Hardware complexity increases.
  - In n-way associative, must search multiple entries
  - Consider the cost of searching a fully associative cache.

Eviction Policy

- So... what to do when the cache is full?
- Under the assumption that all memory accesses must first go through the cache, this means that we need to throw something out of the cache.
- How do we decide what to throw out?

Eviction Policy

- Random
  - Simple
  - As likely to evict a useful block as it is to throw out a useless one.
- LRU (Least Recently Used)
  - “The block that has been used least recently is the one that is least likely to be used again” (temporal locality)
  - Complicated and costly to do perfectly accurately (requires updating every block in set in access)

Eviction Policy

- FIFO (First in First out):
  - Approximates LRU with less complexity.
- MRU (Most recently used):
  - Extremely simple and fast.
  - Not useful if you're trying to utilize locality.
- NMRU (Not most recently used):
  - Randomly evict any block that's not the most recently used.
  - “Approximates” LRU.

Write-Hit Policy (Write Policy)

- What do we do when we write to memory and the address in question is in the cache?
- Write through
  - For each write-hit, write to the cache and ALSO to the backing memory.
- Write back
  - For each write-hit, write only to the cache but mark the cache block as dirty. Then if that dirty block is evicted, write it back to memory

Write-Hit Policy (Write Policy)

- Benefits of Write-Through?

Write-Hit Policy (Write Policy)

- Benefits of Write-Through?
  - Memory and cache are always synchronized and consistent.
- Downsides?

Write-Hit Policy (Write Policy)

- Benefits of Write-Through?
  - Memory and cache are always synchronized and consistent.
- Downsides?
  - Have to incur that crazy penalty every single time you write.
  - Or maybe not? A write buffer can be used in conjunction with a write-through cache. Write to the write buffer after which the processor is allowed to continue while the buffer writes to main memory.

Write-Hit Policy (Write Policy)

- Benefits of Write-Back?

Write-Hit Policy (Write Policy)

- Benefits of Write-Back?
  - Write-through requires 1 write to memory for every store instruction. Write-back effectively collects all of the writes to a particular block and needs to perform one write to memory. This is much better compared to write through if you're frequently writing to the same block.
- Downsides?

Write-Hit Policy (Write Policy)

- Downsides?
  - Write-through pays the cost of writing to memory for stores, but write-back may mean paying the cost of writing to memory for loads.
  - Load performance may be more critical.

Write-Miss Policy

- What do we do when we perform a write and the block ISN'T in the cache?
- Write allocate
  - Pull the block from memory into the cache and then perform a write-hit (what happens afterwards depends on the write-hit policy).
- Write no-allocate (write around)
  - Write only to memory, don't pull the block into the cache.

Write-Miss Policy

- Benefits of using write-allocate:

Write-Miss Policy

- Benefits of using write-allocate:
  - Pulling the block into the cache could potentially help future accesses to memory (ex. storing into sequential elements of an array, repeatedly storing and loading due to register spilling)
- On the other hand...

Write-Miss Policy

- Benefits of using write-allocate:
  - Pulling the block into the cache could potentially help future accesses to memory (ex. storing into sequential elements of an array, repeatedly storing and loading due to register spilling)
- On the other hand...
  - This is only beneficial if there is store locality, otherwise you'd be needlessly polluting the cache with a block of memory that you may not access again anytime soon.
  - The act of allocating takes time.

Write-Miss Policy

- Benefits of using write-no allocate:
  - If a particular set of stores don't have much locality, you can save time by using a write-no allocate.

Caches: Measuring Performance

- ET = IC * CPI * CT
- Previously, we used an idealized CPI that did not take data hazards, control hazards, or memory stalls into account.
- Now we need a more accurate measure of CPI, a Total CPI.

Caches: Measuring Performance

- TotalCPI = BaseCPI + MemCPI
- BaseCPI = PeakCPI + Data Hazard Stalls + Control Hazard Stalls
- Data Hazard Stalls = (frequency of data hazards) * (stall penalty of data hazard)
- Control Hazard Stalls = (frequency of failed branch predictions) * (branch misprediction penalty)

Caches: Measuring Performance

- MemCPI = Instruction Memory Stalls (IM Stalls) + Data Memory Stalls (DM Stalls)
- IM Stalls = (Frequency of Instruction Cache (I$) misses) * penalty
- DM Stalls = (Frequency of memory instructions) * (Frequency of D$ misses) * penalty
- Note that with IM stalls, every instruction must access the IM cache. However, with DM stalls, only certain instructions require the cache.

Caches: Measuring Performance

- AMAT = Average Memory Access Time
- = Time for a cache hit + Miss rate * Miss Penalty

Caches: Measuring Performance

- EX:
  - BaseCPI = 1
  - Write-back, write-allocate
  - Memory access: 400 cycles
  - I$ Miss Rate: 2%
  - D$ Miss Rate: 10%
  - Assume cache is full.
  - 25% of data cache blocks are dirty
  - 40% of instructions are loads/stores

Caches: Measuring Performance

- EX:
  - TotalCPI = BaseCPI + MemCPI
  - MemCPI = (% of I$ miss) * penalty + (% of memory instructions) * (% of D$ miss) * d_penalty.
  - d_penalty = 400?
  - Since write-back, have to consider that evicting a block may mean an additional memory operation. 25% are dirty
  - d_penalty = 400 + .25*400

Caches: Measuring Performance

- EX:
  - TotalCPI = BaseCPI + MemCPI
  - MemCPI = (% of I$ miss) * penalty + (% of memory instructions) * (% of D$ miss) * d_penalty.
  - d_penalty = 400 + .25*400
  - MemCPI = .02 * 400 + .4*.1*(400 + .25*400)
  - = 8 + .04*(500)
  - = 28

Caches: Measuring Performance

- EX Multi-level caches:
  - BaseCPI = 1
  - Write-back, write-allocate, don't worry about dirty writes
  - Memory access: 200 cycles
  - L1, single cycle access:
    - I$ Miss Rate: 20%
    - D$ Miss Rate: 10%
  - L2: Shared miss rate: 5%, 10 cycles to access
  - 10% of instructions are stores
  - 50% of instructions are load
  - 20% of instructions are branches

Caches: Measuring Performance

- EX Multi-level caches:
  - Branches accurate 90% of the time
  - Branches resolved in EX (that sounds familiar)
  - Data Hazard: Load-use hazard, one stall.
  - 50% of loads are immediately followed by a dependent

Caches: Measuring Performance

- TotalCPI = BaseCPI + MemCPI
- BaseCPI = PeakCPI + Data Hazard Stalls + Control Hazard Stalls
- Data Hazard Stall = .5 (% of insns that are loads) * .5 (% of loads that require a stall) * 1 (penalty) = .25
- Control Hazard Stalls = .2 (% of insns that are branches) * .1 (% of branches that are mis-predicted) * 2 (penalty) = .04

Caches: Measuring Performance

- MemCPI = Instruction Memory Stalls (IM Stalls) + Data Memory Stalls (DM Stalls)
- IM Stalls = .2 (% of I$ misses) * (10 (cycles needed to access L2 cache) + .05 (% of L2 cache misses) * 200 (cycles needed to access memory)) = 4
- DM Stalls = (.1 + .5) (% of insns that are lw/sw) * .1 (% of D$ misses) * (10 (cycles needed to access L2 cache) + .05 (% of L2 cache misses) * 200 (cycles needed to access memory)) = 1.2
- TotalCPI = 1 + .25 + .04 + 4 + 1.2 = 6.49
- This is not the CPI of 1 that we were promised!

End of

The Eighth Week

-Two Weeks Remain-