◊ **Pipeline Stages of LC2K**

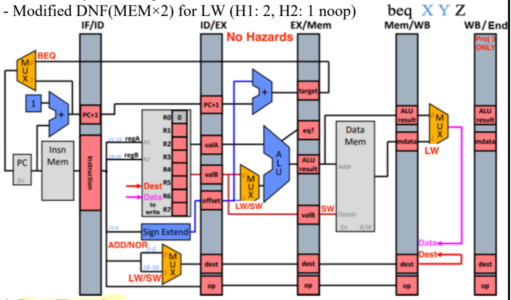
	1	2	3	4	5
ADD/NOR	RIM	**RFR (Reg are read)**	ALU	Pass result to next stage	**WB to reg file**
LW	RIM	RFR	ALU (TargetAddr)	**Read MEM**	**WB to reg file**
SW	RIM	RFR	ALU (TargetAddr)	**WB to MEM**	Nth
BEQ	**RIM (fetch)**	RFR	ALU (TargetAddr & Eq)	**Send TargetAddr to PC if Eq.**	Nth

◊ **Miscellaneous**

- Start at blue and look for the reds above it for DH
- writeData for ADD, NOR: ALU_{res}, LW: Mem[ALU]
- *.fill X* will get decoded to *add 0 0 X* in a pipeline
- 1 instr away means top's at ID/EX and bottom's at IF/ID
- Modified DNF(MEM×2) for LW (H1: 2, H2: 1 noop)

Read Write Ports

add	X	Y	Z
nor	X	Y	Z
lw	X	Y	Z
sw	X	Y	Z
beq	X	Y	Z

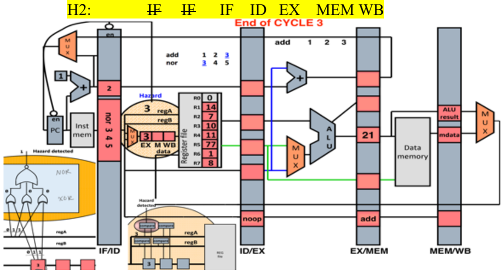


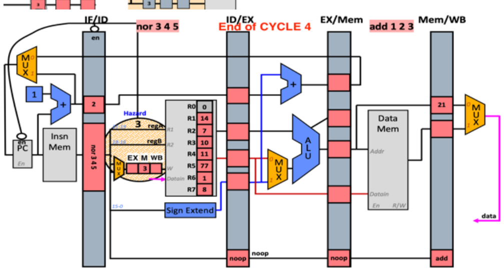
◊ **Data Hazards**

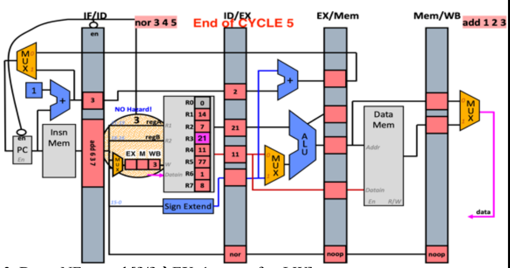
1. Avoid [Insert noops by compiler]
2. DetectNStall [MEM/**WB**→ID, H1: 2noop, H2: H1–1]

beq → EX MEM

Add,Nor,LW: IF ID EX MEM WB
H1: IF ~~ID~~ ~~ID~~ ID EX MEM WB
H2: ~~IF~~ ~~IF~~ IF ID EX MEM WB

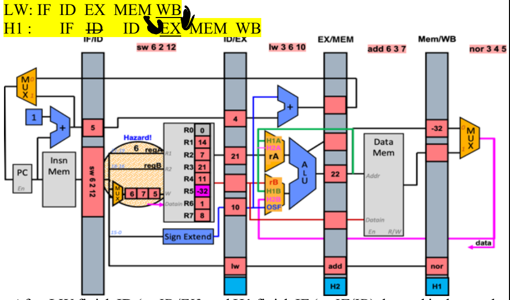






3. DetectNForward [?/?→EX, 1 noop after LW]

LW: IF ID EX MEM WB
H1 : IF ~~ID~~ ID EX MEM WB



- After LW finish ID (*at ID/EX*) and H1 finish IF (*at IF/ID*), hazard is detected at the start of next cycle, so 1 noop is passed ontop of H1 *at IF/ID*. During the remaining cycle, LW finish EX(flow to EX/MEM) while the noop finish ID(flow to ID/EX). **EndofCycle: [IF/ID:H1, ID/EX:noop, EX/MEM: LW]** (All three just proceed to next pipeline register in next cycle)
- At start of next cycle, *wdata* from LW at MEM/WB gets routed back to input MUX at EX, where stale value at ID/EX is another input (control to select prev. one)

ADD, NOR: (0 noop) [H1: EX/**MEM**→EX, H2: MEM/**WB**→EX]
LW: (1 noop) [H1: MEM/**WB**→EX]

Data H	Avoid	DNS	DNF
Pros	Less Hardware Imple.	Backward Compatibility	
	CPI very close to 1	↓ fetching noops(MemAccess), produce smaller executables	Best Perf. among 3
Cons	Backward Incompatibility, Program Size ↑ as noops are inserted (i$ HitRate↓)	CPI ↑ whenever hazard is detected	Higher Hardware Cost (New Datapath, MUX)

byte = 8 bits   word = 2 bytes

◊ **Control Hazards**(MEM/**WB**→IF)

1. DetectNStall [MEM/**WB**→IF, 3 noop whenver BEQ is detected]
- BEQ at IF/ID and nextinstr. is ready to be fetched. At the start of cycle, hazard is detected, so 1 noop is inserted at IF/ID while BEQ flow to ID/EX
- BEQ flow to EX/MEM; old noop flow from IF/ID to ID/EX, 1 more new noop is inserted at IF/ID
- TargetAddr is figured out and get routed from EX/MEM back to PC; BEQ flow to MEM/WB; two old noops keeps propagating; 1 more new noop is inserted at IF/ID
- [IF/ID, ID/EX, EX/MEM: noop, MEM/WB: BEQ] and PC finally can start fetching the correct TargetAddr

2. SpeculateNSquash [3 noops if incorrect]
- Keep fetching and passing instr. that are below BEQ in pipe trace until BEQ reached EX/MEM and is actually taken
- Then at the start of cycle, targetAddr get routed back to PC; BEQ flow to MEM/WB; 3 noops are simultaneously inserted at IF/ID, ID/EX, EX/MEM. (Fetch taken path in next cycle)

Detect & Stall: Branch Taken

beq 6 7 1	IF	ID	EX	MEM	WB				
1		~~IF~~	~~IF~~	~~IF~~					
2					IF	ID	EX	MEM	WB
beq 6 7 2	IF	ID	EX	MEM	WB				
1		~~IF~~	~~IF~~	~~IF~~					
2			~~IF~~	~~IF~~					
3					IF	ID	EX	MEM	WB
beq 6 7 3	IF	ID	EX	MEM	WB				
1		~~IF~~	~~IF~~	~~IF~~					
2			~~IF~~	~~IF~~					
3				~~IF~~					
4					IF	ID	EX	MEM	WB
beq 6 7 4	IF	ID	EX	MEM	WB				
1		~~IF~~	~~IF~~	~~IF~~					
2			~~IF~~	~~IF~~					
3				~~IF~~					
4									
5					IF	ID	EX	MEM	WB

Detect & Stall: Branch Not Taken

beq 6 7 1	IF	ID	EX	MEM	WB				
1		~~IF~~	~~IF~~	~~IF~~	IF	ID	EX	MEM	WB
2						IF	ID	EX	MEM

Speculate & Squash: Branch Taken

beq 6 7 1	IF	ID	EX	MEM	WB				
1		~~IF~~	~~ID~~	~~EX~~					
2			~~IF~~	~~ID~~	IF	ID	EX	MEM	WB
3				~~IF~~	---	IF	ID	EX	MEM
4							IF	ID	EX
beq 6 7 2	IF	ID	EX	MEM	WB				
1		~~IF~~	~~ID~~	~~EX~~					
2			~~IF~~	~~ID~~					
3				~~IF~~	IF	ID	EX	MEM	WB
4						IF	ID	EX	MEM
beq 6 7 3	IF	ID	EX	MEM	WB				
1		~~IF~~	~~ID~~	~~EX~~					
2			~~IF~~	~~ID~~					
3				~~IF~~					
4					IF	ID	EX	MEM	WB
5						IF	ID	EX	MEM
beq 6 7 4	IF	ID	EX	MEM	WB				
1		~~IF~~	~~ID~~	~~EX~~					
2			~~IF~~	~~ID~~					
3				~~IF~~					
4									
5					IF	ID	EX	MEM	WB

for (int i = 0; i < c; ++i)

```
loop: CBZ Xi, end    // c * N + 1 * T
      SUBI Xi, Xi, #1
      B loop         // c * T
end:

loop: SUBI Xi, Xi, #1
      CBNZ Xi, loop  // (c - 1) * T + 1 * N
end:
```

Control H	Avoid	DNS	SNS
Pros	Less Hardware Imple.	Backward Compatibility	
		Easy to program	Best Perf. among 3
Cons	Impractical	Still low Perf. as CPI ↑ once c beq	Higher Hardware Cost

◊ **Branch Prediction** (**Local/Global**)

1-bit Last-Time Predictor: Start with given T/NT, copy the Global/Local Sequence – 1 to the rest

2-bit Counter Predictor: Start with given T/NT, check upward to determine next T/NT (ST/WT/WNT/SNT) and write it as the next (↑ ↘)

Summary on **Regular/Lecture** Pipeline:

Data Hazards (RAW within 2 instr. window)	DNS: Insert noops so that WB→ID on same col. DNF: 1 noop ONLY when LW-Use case
Control Hazards (every branch instr.)	DNS: 3 noops whenever see BEQ SNS: 0 if correct, else 3 noops

High Level Performance Calculation

- **+(#stages-1) IS NEEDED when #instr. is FINITE**
- If didn't say #instructions, assume it's infinte, so no nd +4
- **#noop= FinishingStage – StartStage – #instr_away**
- For DNF_{data}, DNS_{con}, SNS_{con}, the FINISHING STAGE (stage that can actually route back) is 1 after the Textbook/Q-given stage, DNS_{data} doesn't nd this bc register file uses internal forwarding.
- **But, Finishing Stage ≠ Stage Allows Data-Fwd. Notice branch isn't "Data", it can be routed from any stage**
- **If FinishingStage < FwdStage, then FinishingStage is the first stage after that which allows data-fwding**
- **Another way to check #noop**
  1. **Look at the $Path_{Act}$: X ↓ Y**
  2. **Put left finger on Y, right on H1: Y+1, H2: Y+2**
  3. **Count #noops needed for right finger to reach X**
  4. **Equivalent to #stages_in_between**

1m = 1000,000
1b = 1000,000,000
1n = 10^{-9}

Exe time= #Instr. ×CPI($\frac{cycle}{instr.}$)×Clock Period(T/$\frac{s}{cycle}$) = #Cycles×ClockPeriod

Clock Frequency f ($\frac{cycle}{s}$): 10 MHz = 10^7 = 100 ns; 100 MHz = 10^8 = 10ns; 1 kHMz =1GHz = 10^9 = 1ns

$\frac{\#instr.}{s}$ (IPS): $\frac{f}{CPI} = \frac{1}{CPI \times T}$, How many % faster = $\frac{IPS\ new}{IPS\ old}$

#Cycle = (#Stage - 1) + #Instr. + #load_stall + #branch_stall

#Instr. includes halt, and is diff. for Taken/NT branch

#stalls_per_memory_access(MA) = $\frac{MA\ Latency}{Clock\ Period}$

CPI = 1 + $\frac{\#Stage - 1}{\#instr.}$ + #DH_stall + #branch_stall = $\frac{Total\ Cycles}{\#instr.}$

DNS_{data}: %ADDNORLW × Σ[%depen. × #noop(2)]
DNF_{data}: %ADDNOR**LW** × Σ[%depen. × #noop(1)]
DNS_{con}: %branch × #noop(3)
SNS_{con}: %branch × [**MissRate × #noop(3)** +%Taken×(1–MissRate)×#noop]

◊ **Cache(SRAM)**

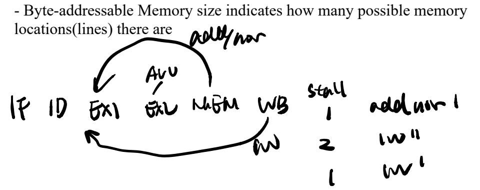
- *Facts*

	SRAM	DRAM	Disks
fast	✓	×	×
cheap	×	✓	✓
	cache	main mem	

- MEM of LC2K: 2^{18}; MIPS: 2^{32}; x86/ARM64: 2^{64}
- SRAM(within processor) & DRAM(outside) are volatile
- DRAM is also a cache as it stores data from hard disk
- Register is physically closest to ALU and has highest access frequency
- FA cache size increase, #tagbits never increase

	Bytes (Binary)	Bytes (Decimal)
1KB	2^{10} = 1024	10^3 = 1,000
1MB	2^{20}	10^6 = 1,000,000
1GB	2^{30}	10^9
1TB	2^{40}	10^{12}

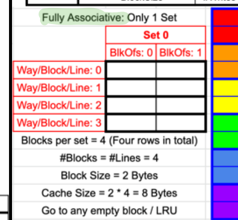
- *Terms*
- Temporal Locality(TL):Reaccess the SAME memory
- Spatial Locality(SL): Re-access NEARBY memory
- Track LRU: Set new accessed blk = 0; ++Remaining; Then replace cache line with highest LRU
- LRU needs update even it's a hit
- Cache line has tag(Addr,CAM)& block(Data,SRAM)
- Cache size is the total amount of data it can store
- Byte-addressable Memory size indicates how many possible memory locations(lines) there are



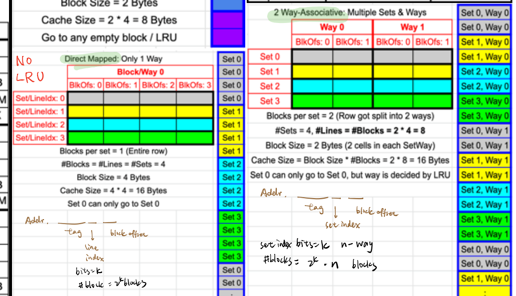
block offset bits = n 2^n - byte block
data = 2^n / block

- *Write Policies*

	MemWritten(MW) & MemRead(MR)			
	Write Back		Write Thru	
AOW	MR = (#Read + #Write) × MissRate × BlockSize	MW = MR × EvictionRate	MR = (#Read + #Write) × MissRate × BlockSize	MW = #Writes × Bytes per write
Not AOW	MR = #Read × MissRate × BlockSize	MW = MR × EvictionRate + #Writes × Bytes per write	MR = #Read × MissRate × BlockSize	



		Write Back	Write Thru
		Fewer #Writes	Write Fewer Bytes
		1 Dirty Bit	0 Dirty Bit
Hit	Write to Cache	ALWAYS	
	Write to Memory	Only when dirty block is evited	ALWAYS
Miss	AllocateOnWrite (AOW)	1. Read Mem & Put on cache 2. Allocate LRU 3. Write to Cache	
		Write to Mem only when dirty block is evited	ALWAYS Write to Mem
	Not AOW	ALWAYS Write to Mem	ALWAYS Write to Mem



	↑Cache Size	↑Block Size	↑Associativity(#ways)
Temporal Locality (#ways)	↑, 有极限 (↓Cap,Conf Miss)	↓ (↓Blocks/Lines)	↑, 有极限 (↓Conflict Miss)
Spatial Locality		↑, 有极限 (↓Compulsory Miss)	
Hit Rate	↑	↑ » ↓	↑
Hit Time	↑	↓	↑
Latency		↑ (load extra data)	
(Tag) Overhead		↓	↑

- *Calculation*
- AddrBits = $\log_2$(MemSize)
- BlkSize = $\frac{Cache\ Size}{\#Blocks} = \frac{Cache\ Size}{\#Lines}$ (Affects SL)
- BlkOfs = $\log_2$(BlkSize) BYTE-addr. WORD-addr: BlkOfs = $\log_2(\frac{BlkSize}{4\ or\ 8})$
- #Blocks = $\frac{Cache\ Size}{\#Blk\ Size}$ = #Blocks per set × #Sets

	FA	SA	DM
#ways	#lines	#ways	1
#sets	1	$\frac{\#lines}{\#ways}$	#lines

#Cache lines ≡ #Blocks; #Ways ≡ #Blocks per set

- LRUBits = $\log_2$(#ways); SetIdxBits = $\log_2$(#sets)
- Total OH = (Tag+Valid(1)+Dirty+LRU) × #lines / **8**
- % OH = $\frac{Total\ OH}{Cache\ Size} = \frac{Tag + Valid + Dirty + LRU}{Block\ Size}$
- *Fully Associative*: AddrBits = TagBits + BlkOfs
- *Set Associative (local LRU per set)*
  - AddrBits = TagBits + SetIdxBits + BlkOfs
- *Direct Mapped (No LRU)*
  - AddrBits = TagBits + LineIdxBits + BlkOfs
  - LineIdxBits = $\log_2$(#Lines) = $\log_2$(#Sets)

Three Types of Cache Misses

1. Compulsory Miss: first reference will always miss
2. Capacity Miss: wouldn't miss with infinite cache
3. Conflict Miss: wouldn't miss with a FA cache

- Simulation: check if tag's alrdy on the block, use SetIdx to determine which set it goes into
- Miss on Inf: Compulsory, New miss on FA:Capacity, New miss on Given: Conflict
- M on Inf(**M_M**): Compulsory, **HMM**: Capacity, **HHM**: Conflict, **__H: NOT ANY MISS**

CPI w cache: 1 + iCache_MissRate×cycles($\frac{Clock\ freq.}{Mem\ Latency}$) + dCache_MissRate × cycles × LoadStore Proportion

◊ **Virtual Memory**

- *Facts*
- 16B: 1 HexDig; 256B: 2HD; 4KB: 3HD, 64KB: 4HD
- Page is the same as Block, so PageSize ~ BlockSize
- Pages are consecutive (no gaps in btw)
- Page Table Entry Contents: **PPN**, valid, dirty, LRU, Access Permission, DRAM or Disk
- Page Table Register holds addr. of beginning of PT
- Can't evict PT whose process is still running (When a new process starts, its PT is pushed on the PhyMem)
- TLB holds PPN instead of the PhyAddr, and it translates simultaneously with cache access
- We have to evict everything in cache if we switch process
- ML PT has higher latency than SL PT(each trans. need #level Mem Access), but it usually takes less space

- *Calculation*

Single-Level PT

- PhyMemSize(PMS); PhyAddr bits = $\log_2$(PMS)
- PA = Phy Page Num(PPN) + Page Ofs
- #PhyPages = $\frac{PMS}{Page\ Size}$ ≠ #VirPages
- VirAddrSpace(VAS) = 2^X for X-bit byte-addressable, VirAddr bits = $\log_2$(VAS)
- VA = Vir Page Num(VPN) + Page Ofs
- #VirPages = #Single-Level table entries(#rows in PT) = $\frac{VAS}{Page\ Size} = 2^{VPN\ digits}$
- Phy PgSize = Vir PgSize; PageOfs = $\log_2$(PageSize)
- Page Table Size= #VirPages × EntrySize(#PPNdigits)

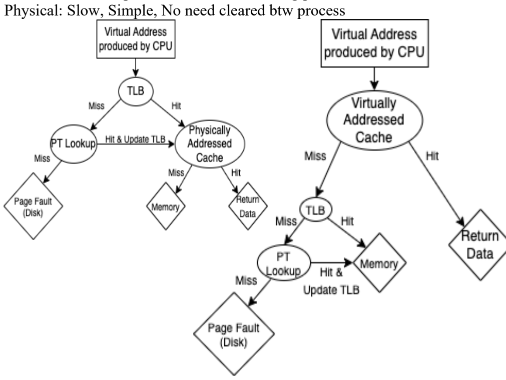
Hierarchical PT

- Ln PT Size = K pages = K × PageSize
- #TableEntry at Ln = $\frac{TableSize_n}{EntrySize_n} = \frac{\#Pages_n \times PageSize}{EntrySize_n}$
- #Ln_bits = log2(#TableEntry_n)   - #L1bits = a; #L2bits = b; #L3bits = c;

For n level:

- MaxPTsize = EntrySize × ($2^a + 2^{a+b} + 2^{a+b+c}$ + …)
- MinPTsize = EntrySize × 2^a, L1 is allocated at start
- SL Ptsize = EntrySize × $2^{a+b+c+\cdots+n}$
- #LeafLevel PT = $\frac{\#Pages}{\#Entries\ at\ leaf} = \frac{\#Pages}{2^{leaf\ level\ bits}}$

Virtual vs Physical Addressed Cache
Virtual: Fast, Complex, Clear when switching process
Physical: Slow, Simple, No need cleared btw process



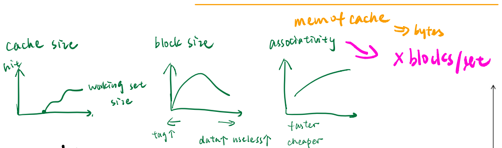
	Virtual Memory (ISA)	
	PageOfs 0	PageOfs 1
Virtual Page 0		
VP 1		
VP 2		
VP 3		
#Virtual Pages = 4		
Page Size = 2 Byte		
Virtual Memory Size = 4 * 2 = 8 Bytes = 2^X, where X is X-bit byte-addressable		

	Physical Memory (DRAM)	
	PageOfs 0	PageOfs 1
Physical Page 0		
PP 1		
#Physical Pages = 2		
Page Size = 2 Byte (same as Virtual)		
Physical Memory Size = 2 * 2 = 4 Bytes		

```
while (opcode(state.MEMWB.instr) != HALT) {
    printState(&state);

    newState = state;
    newState.cycles++;

    /* --------------------- IF stage --------------------- */

    newState.IFID.instr = state.instrMem[state.pc];
    newState.pc = state.pc +1;
    newState.IFID.pcPlus1 = state.pc +1;


    /* --------------------- ID stage --------------------- */
    newState.IDEX.instr = state.IFID.instr;
    newState.IDEX.pcPlus1 = state.IFID.pcPlus1;
    //no stall
    if (opcode(newState.IDEX.instr)!=NOOP) {
        newState.IDEX.readRegA = state.reg[field0(newState.IDEX.instr)];
        newState.IDEX.readRegB = state.reg[field1(newState.IDEX.instr)];
        if(opcode(newState.IDEX.instr)!= ADD && opcode(newState.IDEX.instr)!= NOR){
            newState.IDEX.offset = convertNum(field2(newState.IDEX.instr));
//lw dependency (prev ID is LW)
if(opcode(state.IDEX.instr) == LW){
    if(opcode(newState.IDEX.instr) == LW){
        if(field0(newState.IDEX.instr) == field1(state.IDEX.instr)){
            newState.IDEX.instr = NOOPINSTRUCTION;
            newState.pc = state.pc;
            newState.IFID = state.IFID;
        }
    }
    else if(opcode(newState.IDEX.instr) != NOOP){
        if(field0(newState.IDEX.instr) == field1(state.IDEX.instr) ||
        field1(newState.IDEX.instr) == field1(state.IDEX.instr)){
            newState.IDEX.instr = NOOPINSTRUCTION;
            newState.pc = state.pc;
            newState.IFID = state.IFID;
        }
    }
}
    int regA = state.IDEX.readRegA;
    int regB = state.IDEX.readRegB;

    //LW dependency
    //immediately
    if(opcode(state.WBEND.instr) == LW){
        if (opcode(newState.EXMEM.instr)!=NOOP){
            if (field0(newState.EXMEM.instr)==field1(state.WBEND.instr))
                regA = state.WBEND.writeData;
            if (opcode(newState.EXMEM.instr)!=LW && field1(newState.EXMEM.instr)==field1(state.WBEND.instr))
                regB = state.WBEND.writeData;
        }
    }

    //ADD dependency
    //immediately
    if(opcode(state.WBEND.instr)== ADD ||opcode(state.WBEND.instr)== NOR){
        if (opcode(newState.EXMEM.instr)!=NOOP){
            if (field0(newState.EXMEM.instr)==field2(state.WBEND.instr))
                regA = state.WBEND.writeData;
            if (opcode(newState.EXMEM.instr)!=LW && field1(newState.EXMEM.instr)==field2(state.WBEND.instr))
                regB = state.WBEND.writeData;
        }
    }
    if (opcode(newState.EXMEM.instr)!=NOOP){
        if (field0(newState.EXMEM.instr)==field1(state.MEMWB.instr)){
            regA = state.MEMWB.writeData;
        }
        if (opcode(newState.EXMEM.instr)!=LW && field1(newState.EXMEM.instr)==field1(state.MEMWB.instr)){
            regB = state.MEMWB.writeData;
        }
    }
}

//1-away
if(opcode(state.MEMWB.instr)== ADD ||opcode(state.MEMWB.instr)== NOR){
    if (opcode(newState.EXMEM.instr)!=NOOP){
        if (field0(newState.EXMEM.instr)==field2(state.MEMWB.instr))
            regA = state.MEMWB.writeData;
        if (opcode(newState.EXMEM.instr)!=LW && field1(newState.EXMEM.instr)==field2(state.MEMWB.instr))
            regB = state.MEMWB.writeData;
    }
}

//2-away
if(opcode(state.EXMEM.instr)==LW){
    if (opcode(newState.EXMEM.instr)!=NOOP){
        if (field0(newState.EXMEM.instr)==field1(state.EXMEM.instr)){
            regA = state.EXMEM.aluResult;
        }
        if (opcode(newState.EXMEM.instr)!=LW && field1(newState.EXMEM.instr)==field1(state.EXMEM.instr)){
            regB = state.EXMEM.aluResult;
        }
    }
}
//2-away
if(opcode(state.EXMEM.instr)== ADD ||opcode(state.EXMEM.instr)== NOR){
    if (opcode(newState.EXMEM.instr)!=NOOP){
        if (field0(newState.EXMEM.instr)==field2(state.EXMEM.instr))
            regA = state.EXMEM.aluResult;
        if (opcode(newState.EXMEM.instr)!=LW && field1(newState.EXMEM.instr)==field2(state.EXMEM.instr))
            regB = state.EXMEM.aluResult;
    }
```

overhead of cache:
÷8 ⇒ bytes
(1+1+ k+n +tag) bits/ block · #blocks

memory cache ⇒ bytes



virtually cache

a. When data is in the data-cache? Ans: **1 cycle**

b. When data must be fetched from memory? Ans: **31 and 91 cycles**

Hit TLB: 1 (dcache/tlb) + 30 (memory)
Miss TLB, hit page table: 1 (dcache/tlb) + 30 (1st level pt) + 30 (2nd level pt) + 30 (memory)

c. When data must be fetched from disk? Ans: **100,031/100,061 cycles**

miss in 1st level page table: 1 (dcache/tlb) + 30 (1st level pt) + 100,000 (disk)
miss in 2nd level page table: 1 (dcache/tlb) + 30 (1st level pt) + 30 (2nd level pt) + 100,000 (disk)

◊ **Leftover from Midterm**

- X-bit Two's Comple. $[-2^{X-1}, 2^{X-1})$ [1: 0001, 0: 0000, -1: 1111, -2: 1110]
  $-A = \sim A + 1 = A$ nor $A + 1 = 0$ nor $A + 1$; A nor $A = -A - 1$
- Check even/odd: use mask -2 (111…0)
  nor -2 X 7, beq 0 7 odd
- Byte Addressable: each address refers to one byte in memory
- Word Addressble: each address refers to one word in memory. It can't access individual bytes in a word (32 bit ISA:4 bytes & 64 bit ISA:8 bytes)
- The PC increment is instr. length instead of 1 (LC2K is bc it's 1 word; if 64 byte then jump to i+64 block)

Optimization	Decrease CPI	Decrease # instructions executed	Decrease clock period
Introducing a cache into a previously cache-less system	○	○	○
Switching from a static "predict-not-taken" branch predictor to a more accurate, dynamic predictor	○	○	○
Converting a single-cycle machine into a 5-stage pipeline	○	○	○
Switching from avoidance to detect-and-forward to handle data hazards	○	○	○

	Read	Write
no cache	4	4
no allocate + write through	32	4
allocate + write-back	32	32

→ miss × dirty
miss
4 = ?
32 = full cache block size

allocate on write — new cache
no allocate on write — no new cache
write-through — send to mem
write-back — send to cache "dirty bit"

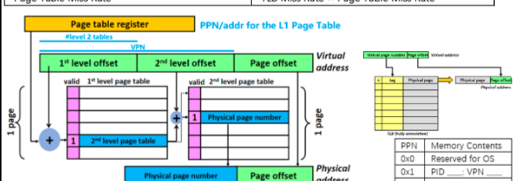
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

	B	KB	MB	GB
1		2^10	2^20	2^30
2		2^11	2^21	2^31
4	2^2	2^12	2^22	2^32
8	2^3	2^13	2^23	2^33
16	2^4	2^14	2^24	2^34
32	2^5	2^15	2^25	2^35
64	2^6	2^16	2^26	2^36
128	2^7	2^17	2^27	
256	2^8	2^18	2^28	
512	2^9	2^19	2^29	

1024 2^{10}

Icache → instruction memory
Dcache → data memory
stalls / cache miss = latency / cycletime
CPI = 1 + data hazard + control hazard + icache → 1 × rate × … + dcache → (lw+sw)% × rate × …

	Virtually Addressed Cache	Physically Address Cache
缓存	L(Cache)	L(TLB) + L(Cache) L(TLB) + #levels × L(Memory) + L(Cache)
内存 TLB H	L(Cache) + L(TLB) + L(Memory)	L(TLB) + L(Cache) + L(Memory)
内存 TLB M	L(Cache) + L(TLB) + (#levels + 1) × L(Memory)	L(TLB) + #levels × L(Memory) + L(Cache) + L(Memory)
Disk	L(C) + L(TLB) + [1, #lvls] × L(Mem) + L(Disk)	L(TLB) + [1, #levels] × L(Mem) + L(Disk)
	Page Fault Rate = TLB Miss Rate × Page Table Miss Rate	Page Fault Rate = Cache Miss Rate × TLB Miss Rate × Page Table Miss Rate



PPN	Memory Contents
0x0	Reserved for OS
0x1	PID ___ : VPN ___
0x2	
0x3	

虚拟内存推演

Virtual Address	VPN	P.O.	Page Fault	PPN	P.O	Physical Address

缓存未命中

Address w/o offset	Tag(SA) bin; hex	Set	INF Compulsory	SA (H/M)	FA	Capaci

	#sets = 1	2	4	stride = 1 block size	2 block sizes	4 block sizes
0	0	0	0	←	←	←
1 block size	0	1	1	←		
2 block sizes	0	0	2	←	←	
3 block sizes	0	1	3	←		
…						
Last block – 3	0	0	0	←	←	←
Last block – 2	0	1	1	←		
Last block - 1	0	0	2	←	←	
Last block	0	1	3	←		

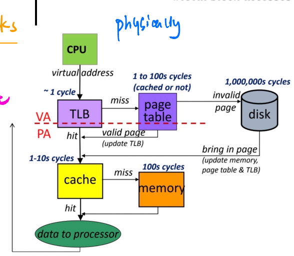
```
if(opcode(newState.EXMEM.instr) != NOOP){
    newState.EXMEM.readRegB = regB;
}

if (opcode(newState.EXMEM.instr) == ADD){
    newState.EXMEM.aluResult = regA + regB;
}
else if(opcode(newState.EXMEM.instr) == NOR){
    newState.EXMEM.aluResult = ~(regA | regB);
}
else if (opcode(newState.EXMEM.instr) == BEQ) {
    newState.EXMEM.eq = (regA == regB);
    newState.EXMEM.branchTarget = state.IDEX.pcPlus1 + state.IDEX.offset;
}
else if (opcode(newState.EXMEM.instr)!= NOOP){
    newState.EXMEM.aluResult = regA + state.IDEX.offset;
}
```

缓存与编程

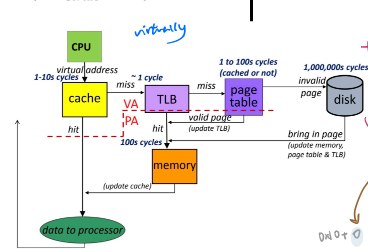
___ improves spatial locality since struct members are more likely to be in the same block.
___ improves temporal locality since fewer evictions will occur and so data will stay in the cache longer.

$$Stride\ Miss\ Rate = \frac{\#missed\ block\ accesses}{\#total\ block\ accesses} \div \min\left(\frac{Cache\ Block\ Size}{stride}, 1\right)$$

physically



virtually



Hardware Component	TLB	Data Cache	Main Memory	Disk
Access Time	1 cycle	1 cycle	30 cycles	100,000 cycles

TLB	
Virtual Page #	Physical Page #

PID 11 Page Table	
Virtual Page #	Physical Page #

Data Cache Tag (Physically Addressed)
0xFBC5

```
/* --------------------- MEM stage --------------------- */
newState.MEMWB.instr = state.EXMEM.instr;

if(opcode(newState.MEMWB.instr)==LW){
    newState.MEMWB.writeData = state.dataMem[state.EXMEM.aluResult];
}
else if(opcode(newState.MEMWB.instr)==SW){
    newState.dataMem[state.EXMEM.aluResult] = state.EXMEM.readRegB;
}
else if(opcode(newState.MEMWB.instr) == BEQ){
    if (state.EXMEM.eq == 1) {
        newState.pc = state.EXMEM.branchTarget;
        newState.IFID.instr = NOOPINSTRUCTION;
        newState.IDEX.instr = NOOPINSTRUCTION;
        newState.EXMEM.instr = NOOPINSTRUCTION;
    }
}
else if(opcode(newState.MEMWB.instr) != NOOP){
    newState.MEMWB.writeData = state.EXMEM.aluResult;
}

/* --------------------- WB stage --------------------- */
newState.WBEND.instr = state.MEMWB.instr;
newState.WBEND.writeData = state.MEMWB.writeData;
if (opcode(newState.WBEND.instr)==LW){
    newState.reg[field1(newState.WBEND.instr)] = newState.WBEND.writeData;
}
else if(opcode(newState.WBEND.instr)== ADD || opcode(newState.WBEND.instr)== NOR){
    newState.reg[field2(newState.WBEND.instr)] = newState.WBEND.writeData;
}
/* ----------------------- END ----------------------- */
state = newState; /* this is the last statement before end of the loop. It marks the end
of the cycle and updates the current state with the values calculated in this cycle */
```

Address Breakdown:

- **Tag Bits**- Bits in our address that we use to refer to the memory block this address corresponds to
- **Set Bits**- Bits in our address that we use to determine what set this address belongs to
- **Block Offset Bits**- Bits in our address that we use to index the value of the memory address from a data block

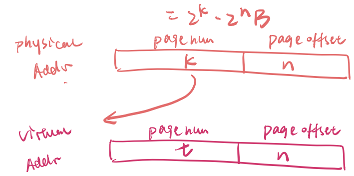
Cache Block/Line Breakdown:

- **Tag** - The label for the memory block currently stored at that cache block
- **Valid Bit** - Whether the cache block is "empty"
- **Dirty Bit** - Whether the cache block needs to be written back to memory upon eviction
- **LRU Bits** - Cache Blocks current LRU ranking within the set
- **Data Block** - The block of data from memory, indexed using the address's block offset bits

page of size 2^nB

pages $= 2^k$ physical
$= 2^t$ virtual

physical memory $= 2^m$B
$= 2^k \cdot 2^n$B



size of each page table entry is > k bits (k+1+1)
total size of virtual page table $= 2^t \cdot$ ↓ ≈ ?