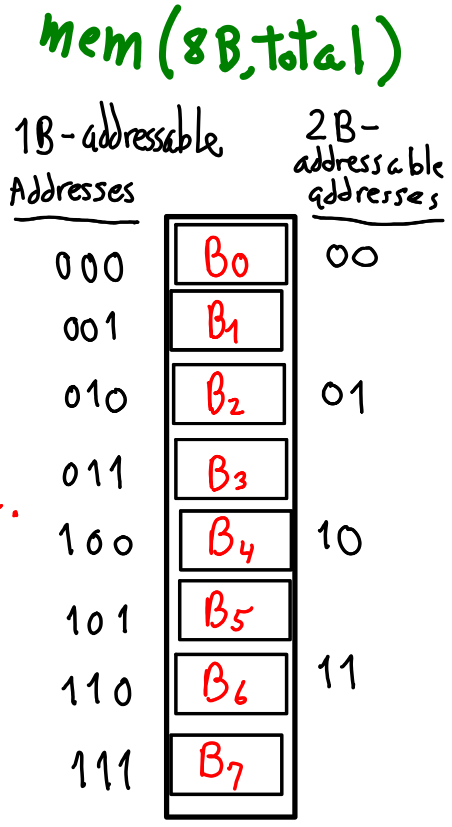
Consider an 128 kB (total data size), four-way set-associative cache with 16 B blocks and LRU block replacement. The cache is physically tagged and indexed. Physical memory is 32MB, byte-addressable, and words are 4 bytes each. Virtual addresses are 32 bits, and pages are 16kB.

Q. How many bits is a physical memory address?

At right is a memory, a sequence of bytes. If a data object can be referenced by a memory address at any byte, the ISA supports byte-addressability. The LC3 can only reference objects at 2B boundaries, it is 2B-addressable. If we add 1 to an address, for kB-addressable, we move forward in memory k bytes.

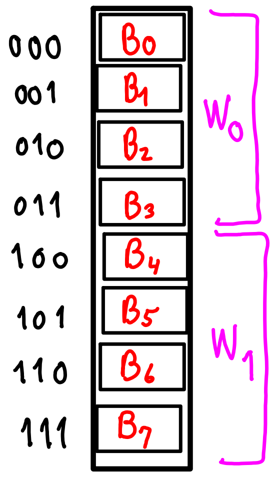


The 8B memory at right requires 3-bit addresses if it is byte-addressable, but only 2-bit addresses if it is 2B-addressable.

Our memory is 32 MB, byte-addressable: we need a different address for every byte: $32M = 2^5 k k = 2^5 2^{10} 2^{10} \Rightarrow 25$ bit addresses.

Q. What is the physical address of the last word of physical memory? What is the biggest (highest) possible virtual address that can be referenced?

An n-byte data object is referenced as the byte at the specified address plus the next (n-1) bytes. For $k = 2^i$ byte objects, the i low address bits are all 0, in a byte-addressable architecture.



In our example above, for 4B words, the low 2 address bits are 0: Word$_0$ starts at address 000, Word$_1$ at 100.

For a 32 MB, byte-addressable memory, the last word starts at byte address (1 1111 1111 1111 1111 1111 1100)$_{\text{binary}}$

or (1 F F F F F C)$_{\text{hex}}$.

The biggest 32-bit virtual address is xFFFFFFFF.

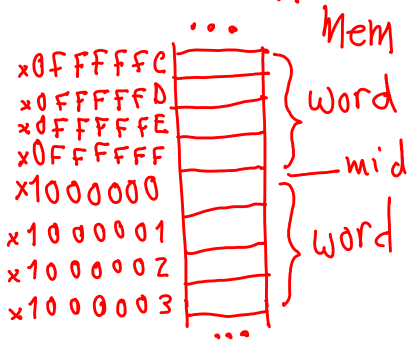
Note that a 4B word could be thought of as a 32-bit binary number. If bytes are bit-ordered $[b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0]$ w/ b_0 the least-significant bit, we can have $[B_3\, B_2\, B_1\, B_0]$ as a 32-bit number w/ b_7 of B_3 the most-significant bit (little-endian) or $[B_0\, B_1\, B_2\, B_3]$, in which case b_7 of B_0 is the most-significant bit (big-endian). The ISA determines which (that can be switched via a bit in a control register in some architectures).

Q. What is the physical address of the middle word of memory? Pick a word such that half (minus 1) of the words are at smaller addresses and half are at bigger addresses. NB--The smallest address has all zero bits, the biggest has all ones.

$\frac{1}{2}$ physical memory $= (\frac{1}{2})\, 32\,MB = \frac{1}{2}\, 2^{25} = 2^{24}$. The first ½ of memory is bytes at address x0000000 through x0FFFFFF, the second ½ is x1000000 – x1FFFFFF.

The last word in the smaller addresses ½ starts at byte x0FFFFFC. Low 2 bits of address are both 0,

$C_{hex} = (1100)_{binary}$



Q. Each cache block has how many words? How many words of physical memory? How many blocks? How many blocks of virtual memory? How many blocks per page? How many pages?

$16\,B$ blocks $\left(\frac{word}{4B}\right) = 4$ words/block.

Memory is $32\,MB \left(\frac{word}{4B}\right) = 2^{25}/2^2 = 2^3 2^{20} = 8\,M$ words.

$8\,M$ words $\left(\frac{block}{4\ words}\right) = 2\,M$ blocks.

$2^{32}\,B$ virtual memory $(block/16\,B) = 2^{28} = 256\,M$ blocks.

$16\,kB/page\,(block/16\,B) = 1\,k$ blocks/page.

$256\,M$ blocks $(page/1\,k\ blocks) = 256\,k$ pages.

Q. What is the address of the first cache block of physical memory? What is the address of the last cache block of memory? What are the addresses of the low words of each block. What are the addresses of the high words of each block?

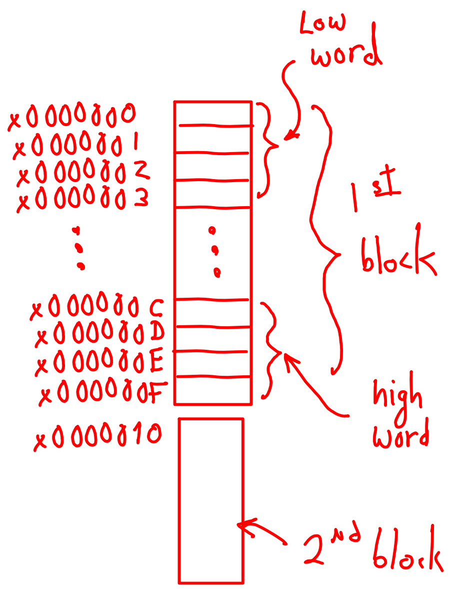
Blocks are 16B; so, last 4 address bits are 0 at block boundaries.

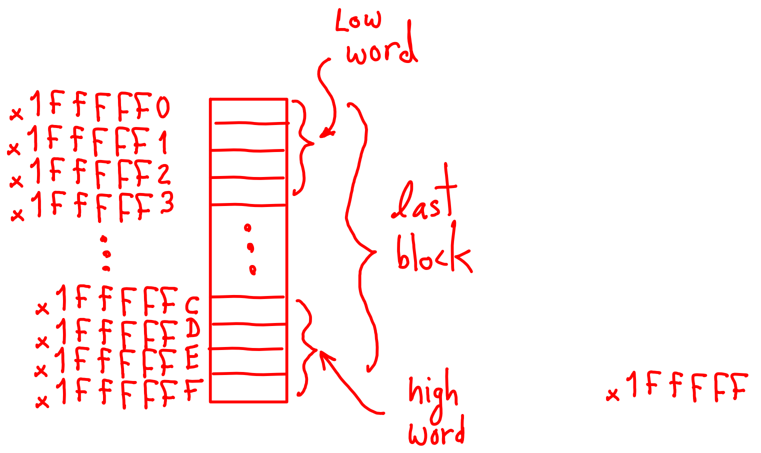
1st Block is at x0000000 (25 bit address), Last is at x1FFFFF0.

The low word of the 1st block is at x0000000.

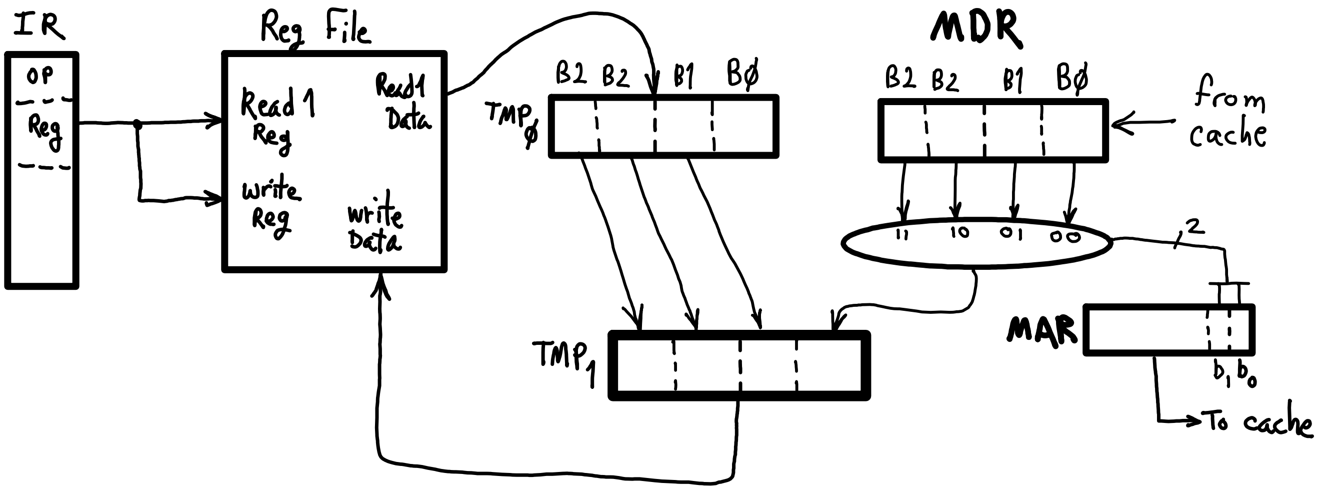
Low word of the last block is at x1FFFFF0

High words are at x000000C and x1FFFFFC.





Suppose the machine's ISA includes a load-byte instruction, LB. LB loads one byte from memory into the low byte of the designated register. The other bytes of the register are unaffected. LB is implemented by fetching the word containing the byte, and then the desired byte is written into the low-byte of the destination register. See diagram below.



The low-order bits of the address select which byte gets written from the MDR. Assume the high-order bits of the instruction register, IR, contain the opcode and the register number. The register field is sent to both the Read1-Reg and Write-Reg select inputs (the same register is read and written). TMP0 and TMP1 are temporary registers in the execution datapath for the LB instruction. Since there is a cache, the "MDR" shown here gets its data from the cache, not directly from the physical memory. The "MAR" contains the address, after translation, that is sent to the cache.

Q. The CPU executes this instruction, LB \$3, 2(\$5). Register \$5 contains xA123456C. The page table entry with the following index (recall, the PT is indexed by page number),

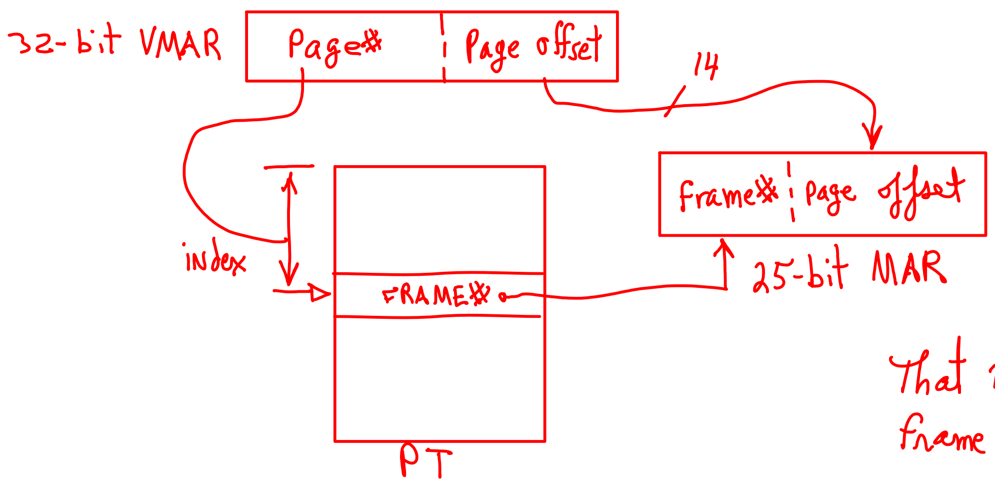
1010 0001 0010 0011 01 (binary)

contains this (physical memory) frame number,

1 1111 1111 00 (binary)

Which byte of which physical word is being accessed by the LB instruction; that is, what is the word's physical address and which byte of that word is referenced? What is the physical address of the cache block containing the requested byte? Which word within the cache block is referenced?

Translation from virtual to physical address is done this way,



Since pages are 16kB, page offset is 14 bits. "Page offset" specifies a byte within a page.

That makes page numbers 32-14=18 bits, frame numbers are 25-14 = 11 bits

The page# (index) above is xA123 followed by $(01)_{binary}$, which is 18 bits.

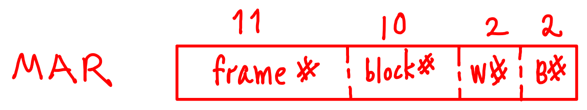
Register \$5 contains x(A123 4 56C), which is 32 bits,

or $\boxed{xA123\ (0100)_{binary}\ x56C}$ the upper 18 bits, xA123 $(01)_{bin}$ are the PT index mentioned above. So, this is translated by replacing those w/ the 11-bit frame# and concatenating the remaining 14-bit page offset $(00)_{bin}$ x56C:

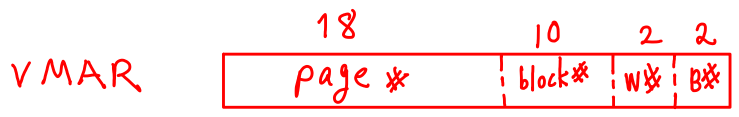
$\boxed{x1FF\ (00)_{binary}\ (00)_{binary}\ x56C}$ which is the 25-bit physical address after translating \$5. The last 4 bits are (1100), and because the last 2 bits are (00), this is a word-aligned address. The offset added to \$5 is 2, which is the byte offset into the word at address x1FF056C, so byte B_2 of that word is referenced. Cache block boundaries end in $(0000)_{bin}$. This block's address is x1FF0560. The word# within the block is 11, or $Word_3$ of that block (the last word).

Q. In the MAR, which address bits specify which byte is referenced within the word? Which bits specify the referenced word within the block? The block within a page? Show as "MAR[k:i]", k > i. Assuming the address before translation is held in a "VMAR" register, do the same for the virtual address bits.

The low 2 bits are the byte offset within a word, B#. As there are 4 words per block, the next 2 bits are the word offset within a block, W#. There are 16 kB/pages, and 16 B/block, or $\left(16\,kB/page\right)\left(\frac{block}{16\,B}\right) = 1k$ blocks within a page. So, the next 10 bits are the block offset within a page, block#. Taken together those constitute the 14-bit byte offset within a page or frame:



MAR[1:0], byte offset within word.
MAR[3:2], The word within a block.
MAR[13:4], block within a page (or frame).
MAR[24:14], physical memory frame number.



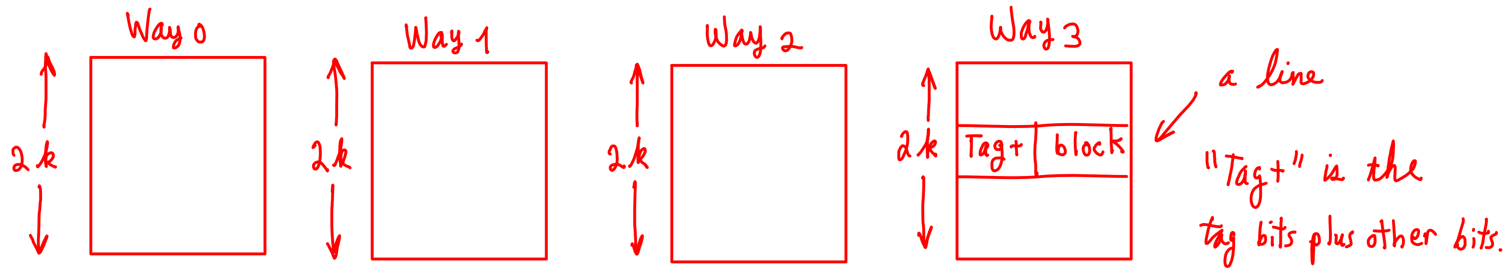
VMAR[1:0], byte offset within word.
VMAR[3:2], The word within a block.
VMAR[13:4], block within a page.
VMAR[31:14], virtual memory page number.

Q. How many cache data blocks does the cache contain, in total? Recall that a cache line contains a data block plus tag bits and any other per-block bits, such as a valid bit and LRU bits. Recalling that the cache has four ways (4-way associativity), how many cache lines in each of its 4 DM cache-line memories?

Total cache data size is $128\,kB \left(\frac{block}{16\,B}\right) = \frac{2^7 \cdot 2^{10}}{2^4}$ blocks $= \left(2^{13} = 8k\right)$ blocks.

$8k$ blocks $\left(\frac{1}{4\ ways}\right) = 2k$ blocks/way : each block in a DM cache-line memory (a "way") corresponds with one cache line

$\rightarrow$ 2k lines per way, i.e. 2k lines per each of the 4 DM cache-line memories.



Q. How many address bits are required to index into a single DM cache-line memory? Which bits of the MAR should be used to address Way-0's cache-line memory? What about indexing for Way-1, Way-2, Way-3?

2k lines → 2^{11} lines → 11-bit index. We can pick any 11 bits in the MAR to use as index bits, but we want to not overlap cache blocks: ie., we want to use the block # (relative to memory)

block#	W#	B#

. Using the low bits of the block number spreads the distance between colliding blocks. So, the MAR layout is

	11	2	2
	index	W#	B#

So, MAR[14:4] are used for indexing into the cache-line memories.
The same 11 bits, MAR[14:4], are used to index all four Ways simultaneously.

Q. How many bits are need per line for tag bits? Which bits of the MAR are used as tag bits? Specify in "MAR[ ]" form. If the cache were virtually tagged, how many tag bits per line?

Physical address are 25 bits; so layout is,

10	11	2	2
Tag	index	W#	B#

10-bit Tag is in MAR[24:15].

Virtual addresses are 32 bits; so, layout is

17	11	2	2
Tag	index	W#	B#

and virtual tags are 17 bits.

Q. Show the bit-field layout of a cache line. Assume 1 valid (v) bit, 2 LRU bits, and 3 cache-coherency (cc) bits per line. How many bits per cache line? Bytes? How many bytes per each way's cache-line memory? What is the cache's non-data storage overhead, as a fraction of data storage? In total, how many bytes of the entire cache are non-data overhead?

Cache line layout is,

10	32	32	32	32	1	2	3
Tag	$word_3$	$word_2$	$word_1$	$word_0$	V	LRU	cc

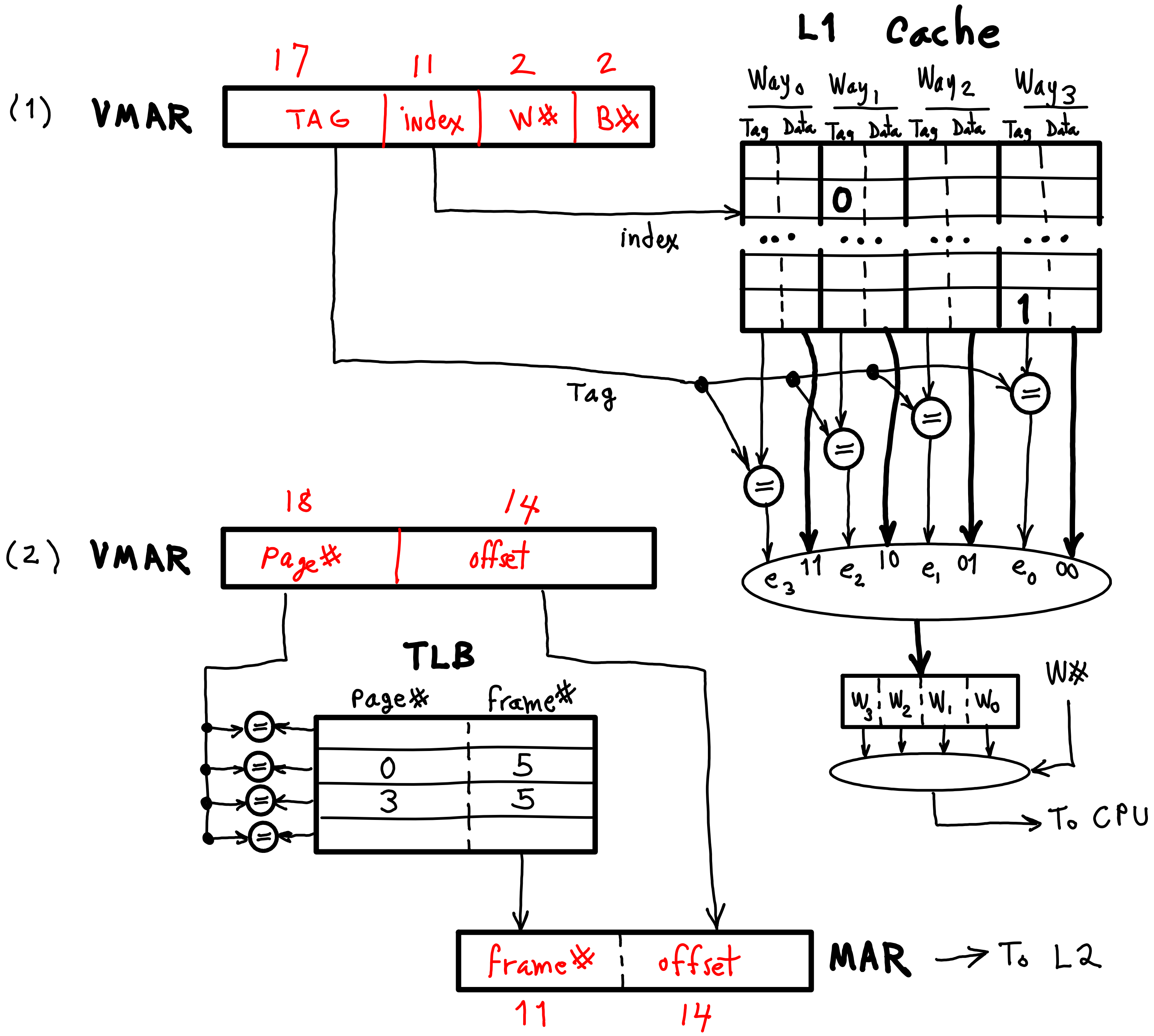
Total # bits is $10 + 4(32) + 6 = 144$ bits/line, which is 18 B.

There are 2k lines → $2^{11}(18\text{ B}) = 2^{10}(36)\text{ B} = 36\text{ kB}$ per way.

Overhead is $2\text{B}/16\text{B} = 1/8$. Total cache data size is $(4\text{ ways})(16\text{B}\cdot 2^{11}/\text{way}) = 128\text{ kB}$

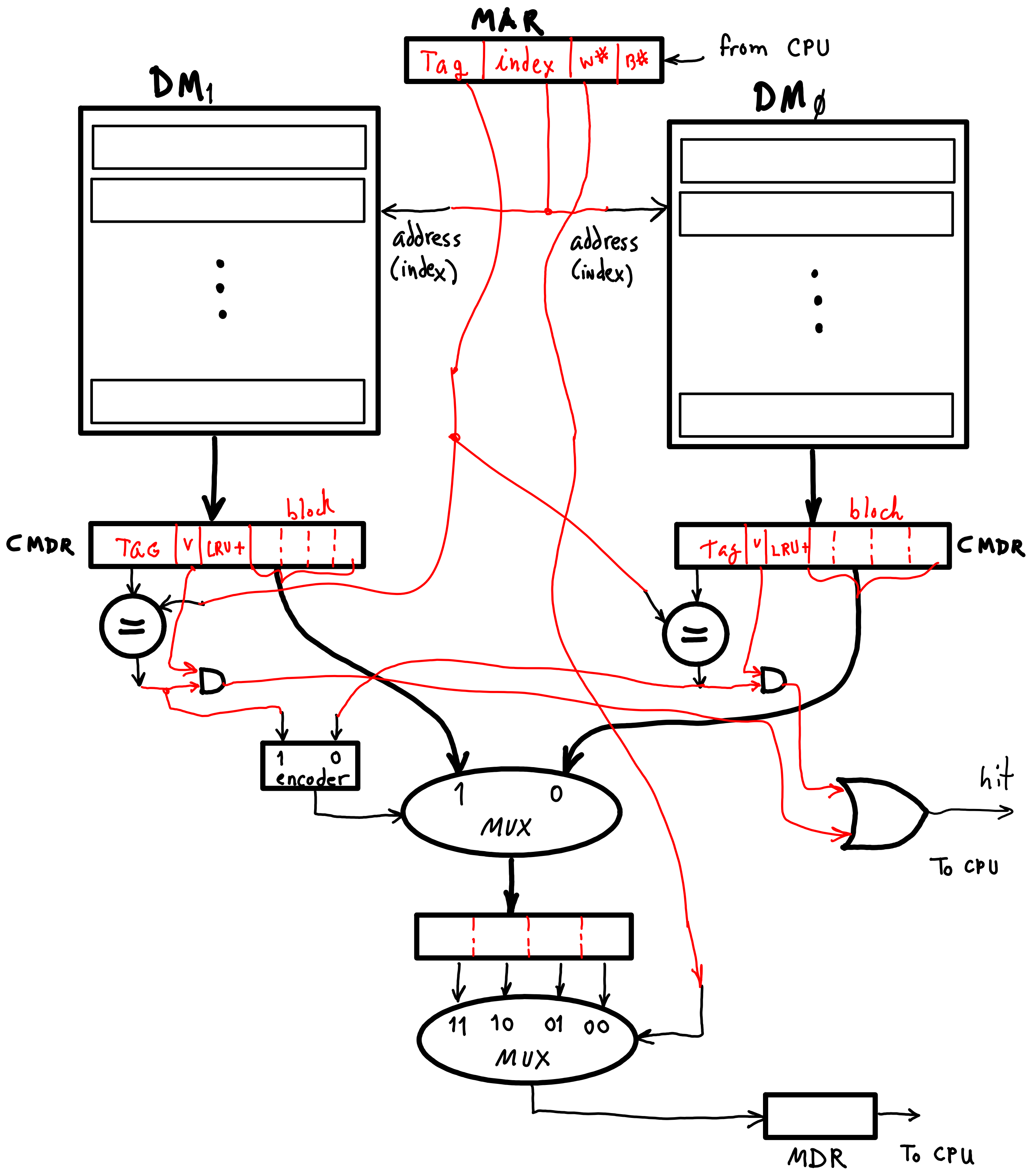
Total overhead is $(1/2^3)\, 2^7\text{ kB} = 16\text{ kB}$.

Q. Suppose the cache is virtually tagged and virtually indexed. That is, both the tag and the DM cache-line memory index come directly from the virtual address before translation. Below is shown two copies of the address register (VMAR) that holds the memory address being referenced in instruction fetch or data access before virtual-to-physical address translation. Show the bit-layout for the VMAR (1) as it pertains to cache tag, cache index, word number, and byte number, and (2) as the same bits pertain to page number and page offset. Show the number of bits used for each field. Show the layout of the MAR also.

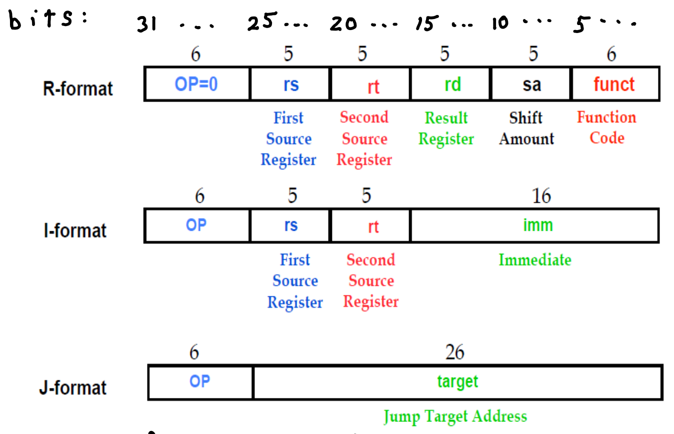


Q. The virtual pages 0 and 3 map to the same physical frame, frame 5. The diagram above indicates a block with virtual tag 0 is in L1. There is also a block in L1 with virtual tag 1. Explain why this could be an example of the synonym problem. Suppose both blocks are dirty.

Q. In the diagram of a 2-way cache shown below, draw the wire connections for DM addressing (indexing), tag comparison and hit detection, and MUX controls for a CPU memory read access. Show the bit-field layout of the MAR and the two DM cache CMDRs (Cache Memory Data Registers). Show the sizes of address, data, and control signals in bits.



Below is shown a 5-stage pipeline architecture (Fetch, Decode, Execute, Memory, and Write-back). Stage registers are the narrow vertical rectangles. At right are instruction formats (high-order bit at the left). Control signals are decoded and then passed through a MUX and written to a pipeline stage register. The control signal destinations are shown as select inputs to datapath MUXs, write-enable signals, and ALU operation control. Branches are taken when the ALU result is 0 (ALU.Zero == 1).

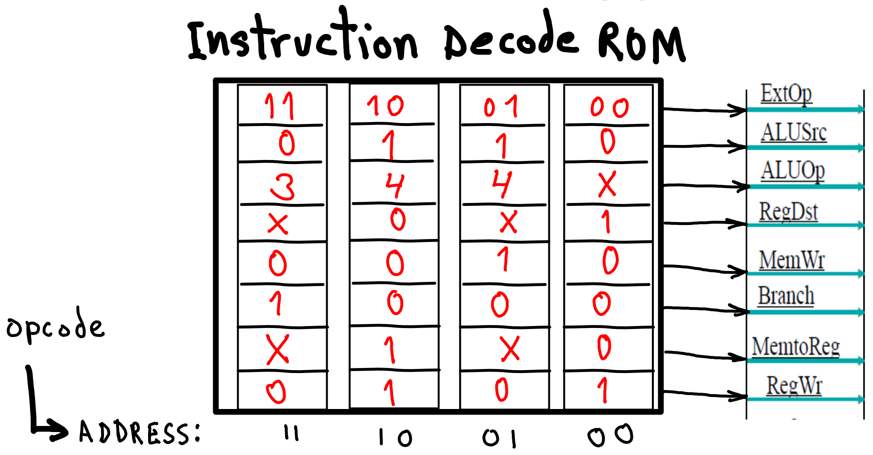


bits: 31 ... 25 ... 20 ... 15 ... 10 ... 5 ...
R-format: OP=0 (6), rs (5), rt (5), rd (5), sa (5), funct (6)
First Source Register / Second Source Register / Result Register / Shift Amount / Function Code
I-format: OP (6), rs (5), rt (5), imm (16)
First Source Register / Second Source Register / Immediate
J-format: OP (6), target (26)
Jump Target Address

Instruction Decoder.
A ROM containing an 8-element word for each opcode.
For simplicity, assume 2-bit opcodes: BR==11, LW=10, SW=01, ALU operations=00.

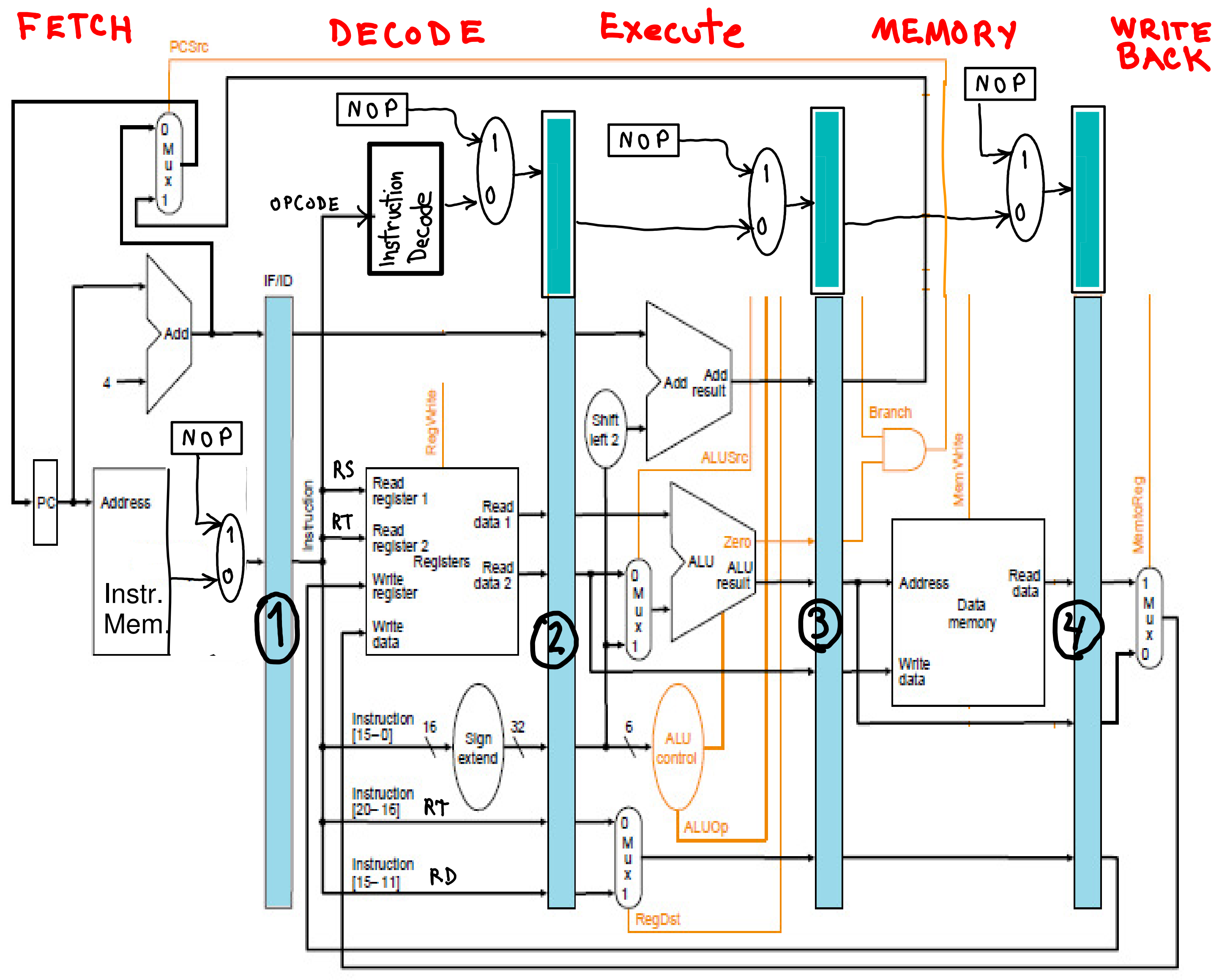
ExtOp:	(use the opcode for this field)
ALUsrc select:	0 = ReadData2, 1 = immediate
ALUOp select:	3 = Subtract, 4 = Add
RegDst select:	0 = RT, 1 = RD
MemWr:	0 = Read, 1 = Write
Branch:	instruction is a branch
MemtoReg select:	0 = ReadData, 1 = ALUresult
RegWr:	0 = No write, 1 = Write

Instruction Decode ROM



ADDRESS:	11	10	01	00
ExtOp	11	10	01	00
ALUSrc	0	1	1	0
ALUOp	3	4	4	X
RegDst	X	0	X	1
MemWr	0	0	1	0
Branch	1	0	0	0
MemtoReg	X	1	X	0
RegWr	0	1	0	1

opcode → ADDRESS:



Q. Identify which pipeline register supplies each datapath control signal. The pipeline registers are numbered 1-4. For example, ALUOp in Execute stage comes from pipeline register 2

ALUsrc in EXECUTE comes from pipe register 2
ALUOp in EXECUTE comes from pipe register 2
RegDst in EXECUTE comes from pipe register 2
MemWr in MEMORY comes from pipe register 3

Branch in MEMORY comes from pipe register 3
MemtoReg in WRITE-BACK comes from pipe register 4
RegWr in DECODE comes from pipe register 4

Q. Fill in the control signal values in the instruction decoder ROM. A branch is taken if two register values are equal. Branch is an I-format instruction. LW uses RT as its destination register field. If a branch were taken, how many NOPs would be generated by the branch hazard detection unit? Which pipeline registers have their instructions nullified (instruction made into a NOP, or control signals set as a NOP)?

3 NOPs are injected into pipe registers 1, 2, and 3 on a taken branch when the BR instruction is in MEMORY.

Q. The instruction opcode is decoded to produce the "Branch" signal shown in the memory stage. This decoding is done in the Decode stage. Is there any reason this decoding couldn't be done in the Memory stage? Would decoding in the Memory stage increase the pipeline's clock rate? Explain.

Assuming the opcode is carried along in the ExtOp field, decoding Branch could be done in Memory stage. The decoding delay is short w.r.t. to the IMem delay, and runs in parallel with that delay. So, decoding would not increase MEMORY delay and CR would not need to be changed.

Consider an instruction mix of 50% ALU, 20% loads, 5% stores, and 25% branch instructions. What would the average CPI be for this CPU? Assume load-use and branch-delay slots cannot be filled by the compiler; 50% of loads incur a load-use bubble; 20% of branches are taken. Ignore memory access considerations; that is, assume a split L1 cache and that every instruction fetch and every data memory access hits in L1. All data forwarding paths are implemented so that stores and ALU instructions do not stall.

A BR instruction that is taken exits the pipeline along with 3 bubbles: $CPI_{BR\text{-}taken} = (1+3)$. Not-taken BRs have $CPI_{BR\text{-}not\text{-}taken} = 1$.
A LW instruction that has a load-use hazard incurs one NOP bubble and has a $CPI_{LW\text{-}hazard} = (1+1)$. LW w/o a hazard has $CPI_{LW} = 1$. All other instructions have $CPI_{other} = 1$.

$$CPI = \frac{\text{cycles for } N \text{ instructions}}{N} =$$

$$\tfrac{1}{N}\Big[(\text{cycles for } BR_{Taken}) + (\text{cycles for } BR_{not\text{-}taken}) + (\text{cycles for } LW_{hazard}) + (\text{cycles for } LW_{no\text{-}hazard}) + (\text{cycles for ALU}) + (\text{cycles for SW})\Big]$$

$$(\text{cycles for } BR_{taken}) = (N_{BR\text{-}taken} \text{ instructions}) \times CPI_{BR\text{-}taken}$$
$$= N(\%\,BR)(\%\,BR_{Taken}) \times (4) = N(\tfrac{1}{4})(\tfrac{1}{5})(4) = N/5$$

$$(\text{cycles for } BR_{not\text{-}taken}) = (N_{BR\text{-}not\text{-}taken}) \times CPI_{BR\text{-}not\text{-}taken}$$
$$= N(\%\,BR)(\%\,BR_{not\text{-}taken})(1) = N(\tfrac{1}{4})(\tfrac{4}{5})(1) = N/5$$

$$(\text{cycles for } LW_{hazard}) = N(\%\,LW)(\%\,LW_{hazard}) \times CPI_{LW\text{-}hazard} = N(\tfrac{1}{5})(\tfrac{1}{2})(2) = N/5$$

$$(\text{cycles for } LW_{no\text{-}hazard}) = N(\%\,LW)(\%\,LW_{no\text{-}hazard}) \times CPI_{LW\text{-}no\text{-}hazard} = N(\tfrac{1}{5})(\tfrac{1}{2})(1) = N/10$$

$$(\text{cycles for ALU}) = N(\%\,ALU) \times (1) = N(\tfrac{1}{2})(1) = N/2$$

$$(\text{cycles for SW}) = N(\%\,SW) \times (1) = N(\tfrac{1}{20})(1) = N/20$$

$$\Rightarrow \overline{CPI} = (\tfrac{1}{N})\Big[N(\tfrac{1}{5} + \tfrac{1}{5} + \tfrac{1}{5} + \tfrac{1}{10} + \tfrac{1}{2} + \tfrac{1}{20})\Big]$$
$$= 25/20 = 5/4$$

Q. A newer version of this CPU moves the BR condition evaluation, target address calculation, and branch logic to the DECODE stage. Additional data fowarding is implemented, so long as it does not increase the clock period. BR now incurs only a single bubble on taken branches. Suppose there is a LW-BR dependency: a LW preceeds a BR and the BR uses LW's destination register as one of its two comparison sources. In which stage would LW's data be available to forward to the BR instruction's equality comparator? How many bubbles must hazard detection insert in this case?

LW cannot forward its data read from memory until it reaches WRITE-BACK: the first possible opportunity to foward the data would be in MEMORY after the data fetch, but doing so would slow down the CR because the critical path would include the data memory and the BR comparator plus branch logic sequentially. So, BR must stall in decode until LW reaches WRITE-BACK: 2 bubbles are inserted into pipe registers 2 and 3.

Q. Suppose 1/5 of the BR instructions in the above job mix are dependent on an immediately preceeding LW. This accounts for a potion of the total load-use hazards mentioned above. The remainder are due to other dependent instructions following a LW. Out of N instructions, how many LW instructions have a depedent BR instruction load-use hazard? How many LW instructions have a load-use hazard due to some other dependent instruction?

There are $N(1/5)$ total LW instructions. There are $(N/4)$ BR instructions, and $1/5$ of these are dependent on an immediately preceeding LW : $(N/4)(1/5) = N/20$ LW instructions have a BR-load-use hazard. the number of LW instructions w/o any hazard is $N(1/5)(1/2) = N/10$. the number of LW instructions with a load-use hazard not due to an immediately preceeding BR is then,

$$(N/5) - (N/20 + N/10) = N(1/20)$$

Q. Given the assumptions in the preceeding questions, what is the speed-up of the new CPU versus the original CPU?

For the new CPU we must recalculate the cycles for LW instructions =

$$(\text{cycles for } LW_{\text{no-hazard}}) + (\text{cycles for } LW_{\text{BR-hazard}}) + (\text{cycles for } LW_{\text{other-hazard}})$$

$$= (N/10)\,CPI_{\text{LW-no-hazard}} + (N/20)\,CPI_{\text{LW-BR-hazard}} + (N/20)\,CPI_{\text{other-hazard}}$$

$$= N\left(\tfrac{1}{10}(1) + \tfrac{1}{20}(1+2) + \tfrac{1}{20}(1+1)\right)$$

$$= N(7/20) \Rightarrow \overline{CPI_{LW}} = 7/20$$

The new $CPI_{\text{BR-Taken}} = (1+1)$

$$(\text{cycles for } BR_{\text{taken}}) = (N_{\text{BR-taken}} \text{ instructions}) \times CPI_{\text{BR-taken}}$$

$$= N(\%\,BR)(\%\,BR_{\text{Taken}}) \times (2) = N(1/4)(1/5)(2) = N/10$$

$$(\text{cycles for } BR_{\text{not-taken}}) = (N_{\text{BR-not-Taken}}) \times CPI_{\text{BR-not-taken}}$$

$$= N(\%\,BR)(\%\,BR_{\text{not-taken}})(1) = N(1/4)(4/5)(1) = N/5$$

$$(\text{cycles for ALU}) = N(\%\,ALU) \times (1) = N(1/2)(1) = N/2$$

$$(\text{cycles for SW}) = N(\%\,SW) \times (1) = N(1/20)(1) = N/20$$

$$(\text{Total cycles for new CPU}) = N(7/20) + N/10 + N/5 + N/2 + N/20$$

$$= \frac{N}{20}(7+2+4+10+1) = N(24/20) = N(6/5)$$

$$\Rightarrow \overline{CPI}_{new} = 6/5$$

$$S_{new-original} = \frac{T_{original}}{T_{new}} = \frac{(\text{Total cycles - original})\ 1/CR}{(\text{Total cycles - new})\ 1/CR} = \frac{N\ \overline{CPI}_{original}}{N\ CPI_{new}}$$

$$= \frac{(5/4)}{(6/5)}$$

$$= 25/24 = 1\,1/24$$

or about 4% faster

Q. Was the improvement in the new CPU's performance worth the cost?

Even though the performance improvement is small, the added cost was nearly nothing: the design was simply re-organized. The only new cost element was the forwarding paths to the BR comparator, a very small component overall. So, yes, it was worth it.