CS M151B Notes

Lecture Notes

1/8 - Week 1

- Computer is an electronic device that can store and retrieve and process data.
- FLOPs are the number of floating point operations per second, they're a measure of performance.
- Instruction Set Architecture is the bridge between the software and hardware.
- Machine Organization delves into the number of cores, what ALU is used, etc
- Physical Design is more of what wires are used, what material, etc.
- Microarchitecture depends on the type of applications you want to run, the physical design you want to use, etc.
    - We want constraints coming from above and below the microarchitecture layer.
- Chip contains a bunch of transistors and wires that implement the machine organization.
- The problem with trying to increase clock frequency is the heat and power problems that come with it. The problem isn't trying to fit everything on the chip.
    - Hard to remove the heat that inevitably comes from the large amount of power that gets used.
- There is a great need for both performance as well as power efficiency.
- Heterogeneous computing means that different areas of the chip handle different programs/threads.
- Big Data characteristics
    - Volume - Scale of data
    - Velocity - Analysis of streaming data
    - Variety - Different forms of data
    - Veracity - Uncertainty of data
- How can we have a data system that is both large as well as fast to access.
    - Processor can crunch data at a fast rate, but how can we get that data onto the chip? The memory right now just isn't fast enough.
- Execution Time = IC x CPI x CT. Overall ET depends on
    - Instruction count (how many are actually executed - based on the actual program)
    - Cycles per instruction (time per operation - based on both software and hardware)
    - Cycle time of the processor (based on physical design).
- Performance is affected by
    - Algorithm - Determines number of operations executed.
    - Programming language, compiler - Determines number of machine instructions that need to be executed per operation.
    - Processor and memory system - Determines how fast instructions are executed.

    - I/O system - Determines how fast I/O operations are executed.
- For performance, we need to be able to design for Moore's Law, use abstraction to simplify design, make the common case fast (b/c that matters a lot), parallelism (trying to do things so that they overlap in time), pipelining (having stages so that you can get parallelism without having to spend a lot of area), prediction (getting work done ahead of time to extract more parallelism), hierarchy of memories (building number of memories b/c we cannot only have one that is both large and fast), and redundancy (numerous components so that you can still proceed even if something fails).
- Application software is the code you write from a high level.
- System software is something more system specific where the actual machine is trying to understand what you've written.
- Hardware encompasses process, memory, I/O controllers.
- Yield is the proportion of working dies per wafer.
    - You'd be able to increase yield by creating a shit ton of dies/components and then turning off the ones that don't work.
- Cost per die = Cost per wafer / (Dies per wafer * Yield)
- Dies per wafer = Wafer area / Die area
- Yield = 1 / (1 + (Defects per area * Die area / 2))^2

1/10 - Week 1

- MIPS and x86 64 are examples of instruction set architectures, which is the agreed upon interface between the software and the hardware.
- The microarchitecture is the implementation of the ISA.
- Two machines that implement the same ISA don't have to have the same machine organization.
    - Basically they don't have to implement it the same way in the hardware but have to have the same functionality specified by ISA.
    - The two machines can be different in their characteristics.
    - "Same ISA, different organizations of that ISA"
- If you have some app, and you want to run it on two different ISAs, you need to have two different compilers, which provides an executable that is in that particular ISA.
- ET = IC x CPI x CT
- The static instruction count can be wildly different from the dynamic count especially if you have for and while loops.
    - You cannot tell the instruction count just based on the executable or the instructions themselves because some will not be executed and some will be executed more than once.
- Instruction count is dynamic, not just static.
- Each ISA has different instruction sets and thus you can take some program X, compile it in ISA 1, compile it in ISA 2, and the two instruction counts could be different.

- Instruction count does not take into account the complexity of implementing the instruction. That comes with CPI. This means that less instructions does not necessarily make the performance better.
- Processor clock is the mechanism for synchronization across different circuit elements.
- Cycle time is the time it takes for one cycle or one tick.
- Clock relates to the complexity of instruction because if you want an instruction to take place within one cycle and one cycle only, then a more complex set of instructions needs a slower clock.
- Single cycle implementation means that whatever instruction you have, it needs to get done in one cycle.
- Pipelined implementation doesn't have that requirement. It has multiple stages and broken into multiple granularities of instructions.
- In pipelined, a decrease in cycle time (faster clock) means that you could have instructions that instead of taking 4 stages, it could take 20 stages instead so there is a drawback in just trying to aggressively scale the clock to be faster and faster.
- If you have a single cycle, then that means that every instruction will take one cycle, and thus the CPI is 1.0. For multicycle, the CPI doesn't have to be 1.0, it is probably more than 1.
- The amount of cycles taken by each instruction is not enough to give you the CPI, you need to know what the program is actually doing so that you can know how many times you're using each instruction.
    - The percentage for each instruction is determined by the application.
    - The amount of cycles per instruction is determined by the microarchitecture. It basically says how the instruction is implemented in hardware.
    - The CPI is the weighted average.
- 2 Apps, Same machine: Different IC and CPI but same CT
- Same app, different compilers, same machine: Different IC and CPI but same CT
- Same app, same compiler, same ISA, different machine architecture: Same IC (because you have exactly the same app/compiler which means the binary is the same), different CPI (cycles that a particular instruction can take might be different because different machine architecture), and different CT (because different microarchitecture).
- Amdahl's Law is to make the common case fast.
    - If you have an execution time of X, and 90% of the time is going to one circuit, and you target that to improve it. You can approach an improved execution time of .1 * X.

<u>1/12 - Week 1</u>

- Chip manufacturing starts with a silicon ingot and then sliced into pattern wafers which are then diced further into dies or chips. Flaws are going to be produced during the process and we want to discard any dies that are affected.
    - Any die that intersects with the flaw has to be discarded along with the incomplete dies on the edges of the circular ingot.

    - If the die is larger, then there is a greater chance that a flaw will intersect with it.
    - The die is the thing that forms the foundation for the chip, onto which the CPU can be built.
- Yield is the percentage of good dies from total number of dies on a wafer.
- Larger dies -> Lower yield (because any single flaw affects a larger area) -> increased cost
- Cost per die = cost per wafer / number of good dies per wafer
  = cost per wafer / (dies per wafer x yield)
- Remember that die area = wafer area / dies per wafer
- After empirical observations, we found that yield is affected by the die area and the number of defects per area.
- Yield = 1 / (1 + (defects per area x die area / 2)^2)
- When thinking about CPU performance, response time is the total time taken to complete some task, while throughput is a measure of how much work gets done (number of tasks) completed per unit time.
    - We care about both, because if we can have parallelism, then the response time won't really get affected (since the time for one task doesn't change) but we'd get an increase in throughput because the computer can do more at the same time.
- Static instruction count is the number of instructions as appeared in memory, while dynamic instruction count refers to the number of instructions that get run while the program is actually executed.
    - We only really care about the dynamic instruction count when thinking about performance.
- CPU Execution Time = IC x CPI x CT
- CPU Execution Time = (instructions / program) x (clock cycles / instruction) x (seconds / clock cycles)
- CPI = Summation of CPI_i x IC_i for all i instructions / Overall IC
- The measure MIPS calculates just the number of instructions that can be done per second. But, it's not a great measure because it doesn't take into account the complexity of the instructions.

1/17 - Week 2

- Class is about the implementation of the CPU.
- Register memory is fast memory that's in the data path of the CPU (it's inside of the CPU and CPU is separate from the memory).
    - Keep in mind that the instructions that the CPU runs are contained in the memory so we need to go to the memory to determine what to do.
- CPU can load or store into the memory and needs to provide an address to determine where to look.
- If you have 4GB of memory, we can think of memory as an array of single byte quantities. Thus there would be 2^32 one byte locations in that large array. To be able to

address those many locations, the address would need to be 32 bits to indicate which one of the locations I want to read out.

- Back then, fewer instructions were good because we didn't have as much memory and it cost a lot to store instructions in memory so we wanted instructions that could do a lot. This is what made CISC architectures popular back then.
- CISC - More instruction types in my ISA but less instructions in memory
    - With CISC the size of the executable would also be smaller, because there would be less instructions.
- RISC - Less instruction types in my ISA but more instructions in memory
- The problem with CISC is that having all these complex instructions makes the creation of the microarchitecture difficult to create. It pushes more of the burden on the hardware.
    - The increased complexity of the microarchitecture may make cycle time worse (because we have complex components in the architecture) or might make cycle per instruction worse.
    - Design time and design verification could also be worse.
    - From the silicon point of view, it could take more material and more space to create these components.
- The problem with RISC is that it pushes more of the burden on the software side (mainly with the compiler) so that the functions can have all the right functionality even with a reduced instruction set.
- CPU needs to: Instruction fetch -> instruction decode
- Some instructions take more space than others (A++ takes one byte while A*B + C*D takes 4 bytes)
    - That is a variable length instruction. The benefit is that you could save space if less complex instructions take less space (You could fit 4 increment operations). They are what you see in a CISC architecture because memory was low. The drawback is that the decoder needs to determine what the 4 bytes that get pulled in represent. Do we have 4 one byte instructions or 1 four byte instruction?
- In a fixed length architecture, every operation takes the same amount of space, but decoding gets increasingly simplified.
    - In RISC, this is preferred since you don't even have a lot of instruction types and complex instructions, so although you may waste some space, it's not as much as the space you'd waste in CISC. The benefit of decoding being easy is worth it.
- **RISC prefers fixed length instructions and CISC prefers variable length instructions.**
- CPU would prefer information to be stored in registers instead of in memory
- Compiler's job is to orchestrate the movement to and from the register file. Has load and store instructions. Load takes from memory and adds to registers. Store takes the modified register value and puts it back into memory.
- Spilling is when too many registers are full and you need to kick some out in order to perform operations on the other values.
- Let's say you compare ADD instructions on RISC vs CISC
    - RISC: Load/Store machine = ADD R0, R1, R2

        - Would have to load A, load B, Add, and then Store C.
        - One drawback could be that there could be a lot of spills.
    - CISC: Reg/Mem machine = M + R -> R
        - Don't have to go through overhead of bringing values into registers. Operands can be from memory or registers. Cycle time would be less. Cycles per instruction could be more because decoding would take more time.
        - Also helps when you're using instructions that only get used like once or twice. Because you don't have to bring values into registers, you don't have to deal with spilling. And anyway, since we're not using that value a whole lot, there's no performance benefit for bringing it into a register.
- Just to summarize, the software side can affect IC and CPI. The microarchitecture can affect cycle time and CPI. The ISA affects IC, CPI, and CT (Affects CT because if you have VL instructions vs FL instructions, then more time might get taken for decoding, and thus we may have to adjust the cycle time needed).
- Operands can come from registers, memory, or can be immediates.
    - Immediates can never change and more importantly, it **comes in the instruction itself**. It's hardcoded in there, instead of the instruction telling you where to look in memory to get the actual value.
    - Trade off involved in putting a value in register and grabbing it vs just encoding it in an instruction as an immediate. Immediate reduces pressure on the register file, won't have to add to a register file. A con is that we need to have an ADDI instruction and there is limited flexibility because that particular ADDI instruction is strictly for that immediate value since you cannot change it since it's encoded into the instruction itself.
- In MIPS, we have 32 registers, each of which has 32 bits. You'd need 5 bits to specify the location (or the register) because there are 32 registers. It's just a location.
- You could also have 2 levels of indirection where you specify 5 bits to look at a register and then inside of that register is a 32 bit memory address, where you then go into that memory location, and pull out the one byte (or 4 contiguous bytes) that it holds.
- In MIPS, we have 32 registers, FL instructions, load/store, and 3 instruction types: R, I, and J.
    - Each instruction type has the same length and each starts with an op code which is all the information that the decoder needs to determine what type of instruction we have. Op code doesn't just tell you which type it is, it also tells you the specific instruction itself (except with R types which has another 6 bits at the end to specify its specific instruction).
- A sample of LW and SW (which are I instructions)
    - LW: R[RT] = M[R[RS] + SE(I)]
    - SW: M[R[RS] + SE(I)] = R[RT]
- With loads and stores, the memory needs a byte address. With things that relate to the program counter, the last two bits are always zero, because it's pulling out 4 contiguous

bytes from the memory. If you're always pulling out at a 4 byte granularity, you need have to encode the last two bits of the literal because you know that it's going to be a 00.

- By default, after we do any instruction, then the program counter will increment by 4.
- For BEQ, if (R[RS] == R[RT]) then PC = PC + 4 + SES(I) else PC = PC + 4
  - SES means shift and extend where I is a 16 bit value, and turn you turn that into 18 bit by adding 00 (going from word granularity to byte granularity) and then sign extend?
- With a jump, we want to fill up a 32 bit PC. We know that the lower 2 bits are 0, so we have 30 bits left, but the jump instruction has 26 bits. We cannot just sign extend. So, we already have an original PC, take the upper 4 bits of that PC, and put that in the front of our 26 bits that are passed into the instruction.
  - 4 bits from original PC + 26 bits passed in + 2 zeros

<u>1/19 - Week 2</u>

- When thinking about creating an ISA, we want to think about the complexity of instructions we have, which entails the length of the instructions, and the formatting of the instruction (how the bits in each instruction are organized) and the number and locations and types of the operands as well as the operator.
- If all instructions only had one operand each, then you'd need to have an accumulator register that kind of stores the intermediate answers until the final result.
- MIPS has a 32 x 32 bit register file which means we have 32 registers which each hold 32 bits each.
- $t0 - $t9 and $s0 - $s7 are temporary registers
- Memory is byte addressable whenever we interact with memory such as getting a value. Words are aligned in memory and are 4 bytes in length, and the address must be a multiple of 4.
- Some MIPS instructions
  - Add $t0, $t1, $t2 means $t0 = $t1 + $t2
  - Sub $t0, $t1, $t2 means $t0 = $t1 - $t2
  - Addi $t0, $t1, 4 means $t0 = $t1 + 4
  - Add $t0, $t1, $zero means $t0 = $t1 (This implements a move instruction functionality)
  - Lw $t0, 16($s1) means we're taking the address provided by the second argument, and putting that in register $t0. It means $t0 = $s1[4]
  - Sw $t0, 48($s1) means we're taking the value in register $t0 and copying that to the address provided by the second argument. It means $s1[12] = $t0.

<u>1/22 - Week 3</u>

- CPU contains the register file and is a separate piece of hardware from the memory.
- Spilling is when you have to kick out a value from the register to make space for other values. If the value in the register was modified, then you have to do a store word to store the modified value back into memory.

    - Either a load and a store or just a load
- RISC is fixed length and we have 3 instruction types (R, I, and J)
- If you make a more complex operation (LAW instead of L and Add), then that could affect cycle time since you now have a more complex operation that could take more clock ticks and so we have more to do and therefore a slower/faster clock might be better.
    - You might want to have slower clock or more stages (with a variable cycle time).
    - With this change, the instruction count will hopefully (if there are load words and adds) decrease, CPI could increase, and cycle time could potentially increase.
- If let's say your op code needed to be 7 bits, then for I and J types, we'd need to decrease the number of bits used for immediate (16 to 15 for example). This means that we have to fit our offset in less bits and thus this might increase the number of instructions you need because you may need more jumps to get to wherever you wanna go. This would increase IC, mainly for BEQs and ADDIs.
    - If you have LW, and you can't fit the displacement, then you need to have a LW plus an ADD.
    - Basically, the instruction count could increase greatly if the opcode is increased
- By extending the opcode, CPI may or may not be affected. It's a mix of application/compiler as well as the architecture.
- CPI would not change with a change in IC if CPI is 4.0, and the additional instructions are also 4 cycles.
- Another way to deal with an extension of the opcode is that we can just have all our instructions be 33 bits, which means that we cannot store as many instructions in our instruction cache, we'll have more misses on the cache, and therefore the CPI would probably increase.
- If you want to increase the size of the register file, that could possibly decrease the instruction count since there is less pressure to spill and thus less spilling (which reduces the need for some load and store words).
    - One side effect is that we'd have to increase the number of bits used to represent the location of the different elements of the register file. This means that you may have to decrease the number of bits in the opcode, which could then increase the instruction count because that means you need to drop some complex instruction types and thus you need to have more instructions to have the same functionality as before and so IC increases.
    - CPI would likely drop because we would have less spills (and thus less loads, adds, stores, etc) and because we would have less complex instructions (since we've lost some bits in the op and func codes).
    - Cycle time could potentially increase because the register file would take longer to access (higher latency) and thus we have more to do and so we can either slow down the clock or break it up into multiple stages.
- **If you have a single cycle datapath, the CPI is going to be 1.** Everything basically has to be done in one tick. The CPI will always be 1, and thus cycle time would be

terrible since you need a slow enough clock to be able to fit all of that latency into one cycle.

- There is an extension to ARM where you can have both 32 bit and 16 bit instructions. The pros to this would be that you can fit more instructions in cache/register file and thus CPI could decrease. The cons are that it would be harder to decode (drawback of variable length overall).
- Side Note: Preservation of operands is a big advantage to 3-address code instead of 2-address code, while 2-address would give you less register pressure.
- One of the benefits of RISC is pipelining and parallelism. RISC helps those because the simpler instructions can be pipelined more.
    - X = a + b + c + d can be parallelized but you had a complex ass instruction that needs 4 operands, then you cannot do anything, but if you had simple instructions then you can do a + b first and then c + d later.
- CISC (x86 in particular) have only 8 registers and thus hella register pressure, but you have variable length which helps fit instructions a bit better, but you have the classic problems of not being able to pipeline or parallelize.
- ARM looks for small instructions and seeks to fuse them together into macro-ops. X86 tries to break up into micro-ops.
- Jump instructions are explicit in that you hardcode a number into the immediate. Jump and link is implicit in that it will do $ra = PC + 4, as opposed to having a register specifier.
    - Jr is different because it can jump to any register. You'd want to have that so that you can jump to any place, but you have to set up the correct registers.
    - Jal and Jr are the same though in that they both jump somewhere, just depends on where they're jumping.

1/24 - Week 3

- Terms
    - Instruction Fetch - Grabs instruction from memory
    - Instruction Decode - Figures out what that instruction is
    - Execution - ALU performs its job
    - Memory
    - Write Back - Write back to the register file
- When we're adding two N bit numbers, you can think of each bit location having 3 inputs: the two numbers and then the carry in.
    - The two outputs are the carry out and the sum.
    - That's what's called a full adder
- A MUX normally takes in the operation which indicates which result we want to actually return.
- We always compute the 3 values no matter what, and then we use the MUX to select which output we want to return.

- When you want to do subtraction, you need to negate one of the input operands (by taking the complement of one of the inputs) AND you have to set the carry in to 1 since it's not enough to just take the complement of it.
- The distinct quality of ripple carry adders is that we connect the carry out to the carry in of the next unit.
    - Thus, when you want to negate a number, set the C0 (carry in to the first unit) to 1 and take the complement of one of your inputs.
- SLT is an R type instruction.
- Basically if R[RS] < R[RT] then R[RD] = 0...01 else R[RD] = 0...00
- To do the comparison, we simply have to do R[RS] - R[RT] and then check the most significant bit of the result which indicates the sign.
- The SET of the most significant bit (on ALU 31) is connected to the LESS of the least significant bit (on ALU 0). This is called our feedback loop.
- The LESS for every other ALU is 0 since we need to have 0's everywhere except for the LSB which can be 1 or 0.
- So to do subtraction, the carry in = 1, Ainvert = 0, Binvert = 1, opcode = 3 (SLT). So we'll be computing A ^ ~B, A V ~B, etc but our opcode is 3 so we don't really gaf about those other 3 values. The SET on the MSB ALU will have the correct A - B value. Based on the SLT implementation, the MUX on the LSB ALU takes that LESS value and returns it since the op code is 3.
- One of the painful parts of the SLT implementation is that there is a lot of latency with the adder chain that has the carry in of one ALU unit depend on the carry out of the previous unit, which is the ripple carry characteristic.
- In a typical full adder where the inputs are A, B, and Cin, and the outputs are Cout and S
    - Then Cout = (A ^ B) + (A ^ Cin) + (B ^ Cin). The gate implementation will have 3 AND gates and 1 OR gate. If each gate has a delay of T, then there will be at least a 2T delay for each full adder.
    - S = (A XOR B) XOR Cin. We can do A XOR B because we already have those quantities ready in the beginning, while the Cin quantity depends on the units before. We only technically need to add a delay of T since we already know that A XOR B will likely have already been computed by that time.
- S at the zeroth bit will be ready at 2T since we have to calculate A XOR B and the carry out will be ready at 2T and then XOR it with Cin. S1 will be ready at 3T. The carry out will be ready at 4T and so.
    - So S at N-1 will be ready at 2TN - 1. And Cout at N-1 will be ready at 2TN.
- The idea behind the carry look ahead adder is that we want to be able to calculate Cin before having to take the value from the Cout of the previous unit. We will be doing more work in total but the work will be overlapped with each other so we'll have parallelism and thus it'll help with execution time.
- If A and B are 0, we know there won't be a carry (Cin = 0). If A and B are 1, then we know there will be a carry (Cin = 1).
    - For the other two cases, we need to compute other values, one called G which calculates A ^ B as well as another called P which calculates A XOR B.

- Generate and propagate calculations don't depend on Cins. G0, P0, G1, P1, G20, and P20 will all be ready at Time T.
    - C1 however, will be ready at 3T. Because it takes T to calculate P and G, takes T to do the AND, and takes T to do the OR. C2 and C3 are 3T as well.
- The delays of different gates can depend on the fan in. The delay for a gate will be KT.
- We can also create 4-bit CLAs and connect them in a ripple carry fashion so that we create a hierarchical CLA.
    - We add another level of carry lookahead logic.
    - We're basically precomputing the carry in to each of the CLAs. With a singular CLA, we do the same thing with individual ALU units.
    - We have 4 bigger generates and propagates coming out of each of the 4 CLAs.
- In order to determine whether a 4-bit CLA has an overall propagate, it needs to have propagated at every single 1 bit unit.
    - P = P0 ^ P1 ^ P2 ^ P3
- In order to determine whether a 4-bit CLA has an overall generate, it's the same except we don't consider the Cin because we don't want to have to wait for that shit.
    - G = G3 + G2 ^ P3 + G1 ^ P2 ^ P3 + G0 ^ P1 ^ P2 ^ P3
- Using the KT assumption, G0 and P0 will be 2T. Palpha will be 6T (2T plus 4T from the 4 operand OR). Galpha will be 10T (2T plus 4T from the highest operand number AND plus 4T from the 4 operand OR).
    - 10T for all the G's at the top level and 6T for all the P's at the top level.
    - C12 will be 17T (10T from G plus 3T from the AND plus 4T from the OR)
    - C16 will be 19T (6T from P plus 5T from the AND plus 3T from somewhere plus 5T from the OR)
    - C15 will be the worse, since the carry in into it will be 17T plus 4T from the AND plus 4T from the OR, so it'll be 25T.
    - S15 will be the worst, since you'll have 25T for Cin plus 2T for the AND with (A XOR B)

1/26 - Week 3

- Conditional Operations
    - Beq $t0, $t1, L1
        - If ($t0 == $t1) then branch to L1
    - Bne $t0, $t1, L1
    - J L1
        - Unconditional jump
    - Slt $t0, $t1, $t2
        - If ($t1 < $t2) then $t0 = 1 else $t0 = 0
- We can use slt instructions in conjunction with the bne or beq instructions where we can check the value of $t0 for a specific value and then branch based on that.
- Procedure call instructions
    - Jal Procedure label

        - Address of the following instruction put in $ra
        - Jump to target
    - Jr $ra
        - Copies $ra to program counter
        - In effect, this means that the next instruction that will be read is $ra.
- 32 bit constants
    - Lui $t0, upper_const
        - This will also zero out the lower 16 bits.
    - Ori $t0, lower_const
- For example, if we want $t0 = 0x30050, then we can do
    - Lui $t0, 3
    - Ori $t0, $t0, 0x50
- Want to implement jgt $s0, $s1, 0x3A015432, which means if $s0 > $s1, jump to that address
    - Slt $t0, $s1, $s0
    - Beq $t0, $zero, 3 (offset specified relative to the next instruction)
    - Lui $t0, 0x3A01
    - Ori $t0, $t0, 0x5432
    - Jr $t0
- Something to note is that if we have to go to a specific address that is 32 bits (like in the above case), we cannot really use a branch since that branch only has space for 16 bits. Therefore, we have to use a jr.
- Branches have 6 bit opcode, 5 bit rs, 5 bit rt, and 16 bit offset. The offset is specified in units of word.
    - Target address = address of instruction + 4 + (4 * offset).
- Jumps are different from offsets in that we have 26 bits to work with. We get the overall address by taking the upper 4 bits of the PC, then put the 26 bits of the address (left shifted by 2), and the 2 zero bits that result from the left shift.
    - Target address = PC (top 4 bits) and 26 bits and then 00
- Cannot use a single jump instruction to get from 0x00000000 to 0x20014924
- Cannot use a single branch instruction to get from 0x1FFFF000 to the same address because the offset is signed and thus you can specify a negative offset.

1/29 - Week 4

- Carry lookahead should have better performance than ripple carry since we're computing values without needing to wait. But we'd also need additional area to handle the additional lookahead logic.
- We also have a carry select adder that also probably will be more efficient but will need more area.
- The idea behind carry select is that we remove the bottleneck of carry in slowing down our adding. The solution is that instead of waiting, we use two adders and compute one

with select as 0 and one with select as 1, and then determine which to output when the actual carry in comes through.
    - Additions will be happening in parallel.
    - Kinda like speculation since we're doing these two additions in the beginning and then deciding later which output we want to have.
- You can use all these components (CLA, HCLA, RC, CSA) in the same 32-bit adder. You can use a CLA for the first 4 bits, and then maybe use 2 HCLA for the next 16 bits (so that you can do the computation, and then when the carry in comes in, then you can output the right number).
    - The addition is done in parallel and you don't need to wait for anything but you do have to wait for the Cin so that you can choose which output to give.
- Creating a 32 bit multiplier
    - Need a 64 bit ALU since our product will be 64 bits.
    - We shift the multiplicand left and we shift the multiplier to the right.
    - Control test is looking to see if the multiplier's LSB is 1, and if so, then add multiplicand to the product.
        - Then, regardless of 0 or 1, we shift the multiplicand register left 1 bit and we shift multiplier register right 1 bit.
- Optimize multiplier
    - Don't need a 64 bit ALU, just need a 32 bit ALU but instead we store the multiplicand in the product, and we need to shift product to the right.
        - Same as shifting the multiplicand to the left.
    - Add upper 32 bits of the product with the multiplicand.
- Booths algo focuses more on shifting than a sequence of additions.
- Booths algo only works when we have a sequence of 1s and the worst case would be 010101…

<u>**1/31 - Week 4 and 2/5 - Week 5 -** Notebook Notes</u>

<u>**2/7 - Week 5**</u>

- Steady state is where there is some instruction at every stage.
- Important to note the maximal delay at every time step.
- The stages in pipelined approach are IF, ID, EX, MEM, and WB.
- With a pipelined approach, the goal is that your CPI will approach 1.0. It won't be there at the beginning because you're still waiting for instructions to fill up the different stages.
- In pipeline, the cycle time will approximately by equal to the maximal delay of one stage, rather than the sum of all stages in a single cycle datapath.
    - The hope is that if you have X latency for a single cycle and then you break it up into n stages, then the CPI = X/n. But in reality the stages will be heterogeneous and thus we are going to be limited by the worst case stage.

- Pipeline latches are going to hold data that we need between stages. This is what allows different stages of the system to run at the same time. The data is available in the latches and thus one stage can work on different data at once.
- Instruction information is stored in the data between IF and ID.
- One of the drawbacks in the pipeline approach is that your cycle time becomes the maximal delay of one stage.
  - The overall latency if you're just doing one instruction is probably better with single cycle instead of pipeline.
- Time taken in pipeline = # of instructions + time it will take to reach the steady state
- Structural hazard is when two instructions are in the same stage and are trying to use the same RF/ALU/MEM, etc.
- For different instructions, they could use different stages, but every instruction needs to go through the same pipeline and it cannot skip ahead in order to avoid structural hazard
- IF/ID latch holds PC+4 as well as the instruction.
- All the control logic is saved in the latches. Contrast that with the single cycle where all the logic just gets fed into the different stages.
- Data hazards are where the output of one instruction influences the next instruction. So that means that you cannot immediately start the next instruction until the previous one is complete.

2/9 - Week 5

- ISA state is any aspect of the ISA that can observed by the programmer.
  - State is anything related to data or control in the ISA.
  - Memory and the register values and instructions
- ISA implementation state cannot be observed by the programmer and has to do with the specific way the ISA is set up (single cycle, multi cycle, control path, etc)
  - Control path bits, intermediate values,

2/14 - Week 6

- RAW hazards refer to reads after writes.
- Main disadvantage of the no-op software approach to solving data hazards is that we lose portability since the software has to know what's happening in the microarchitecture implementation.
- With no-op injection, the CPI will stay close to 1, and the CT will stay the same, but the instruction count would go up.
- The compiler can also try to reorder the code (this is called code motion) so that you have independent instructions right after one another so that we can decrease the number of no-ops we would need to add.
- So, when the 2nd instruction is in the ID phase, we want it to wait until that first instruction gets to the WB phase. We know whether to wait or not by comparing RS or RT of the instruction in the ID stage with Destination register that is in the EX stage.
  - If (ID.RS == EX.DEST) and (EX.REGWRITE == 1)

    - If (ID.RT == EX.DEST) and (EX.REGWRITE == 1)
        - ADD $t0,_,_ …. , OR _, $t0, $t1
    - If (ID.RS == MEM.DEST) and (MEM.REGWRITE == 1)
    - If (ID.RT == MEM.DEST) and (MEM.REGWRITE == 1)
    - If any of the above 4 are true, then we have a potential data hazard
- If we detect a potential hazard, then we want to bubble. We want the future instructions to just stay where they are.
- As soon as we want to stall, we set all the control bits to zero. The throttling is determined by if we set all the controls to zero, or set them back to normal.
    - This turns the instruction into a no-op where we do all the operations but nothing is getting saved anywhere.
- There will be a hazard detection unit, and then outputting a stall control signal that will get connected to the different MUXes so that they can turn the control signals into 0's if need by.
- Only write to IF/ID if you're NOT going to stall and that's why we have the invert of the stall control signal.
- Hardware bubbling/stalling will not increase IC, but CPI will increase and a possible increase in CT (if the hazard detection unit has the critical path).
- Key thing to note with pipelining is that the individual instruction latency gets longer (increases to 5 cycles), but the throughput increases since we can now get different instructions into the pipeline at the same time.
- Forwarding makes use of the fact that the result of the first instruction is actually available at the end of EX. The 2nd instruction then can get that value before its EX stage. We don't want the first instruction to have to wait until WB stage to write that value back.
- In forwarding, you have a couple different situations. You can forward a value from the Ex/Mem latch to the next instruction or you can forward it from the Mem/Wb latch. When the future instruction is lined up with the first instruction which is in the Wb portion, we don't need forwarding since the true value with get written in the first half.
    - The choice that gets made depends on what stage the first instruction is on, as well as what the future ones are at.
- To set that control, we need to look
    - If (Ex/Mem.RegWrite and Ex/Mem.RegisterRd is not 0) and (Ex/Mem.RegisterRd == ID/Ex.RegisterRs)
        - ForwardA = 10 which means the forward comes in from the Ex/Mem latch
        - Basically the 10 represents that we want the value from Ex/Mem section
    - If (Mem/Wb.RegWrite and Mem/Wb.RegisterRd is not 0) and (Check that the previous equation is NOT true) and (Mem/Wb.RegisterRd == ID/Ex.RegisterRs)
        - ForwardA = 01 which means the forward comes in from the Mem/Wb latch
        - Basically the 01 represents that we want the value from Mem/Wb section
    - We also do the same for the ForwardB signal except it is dealing with ID/EX.RegisterRt

    - If the first equation is true then we always go with that since the value in Ex/Mem will have precedence over the value in Mem/Wb
    - The 3 things we always check are
        - RegWrite in Ex/Mem or Mem/Wb
        - The destination registers from Ex/Mem or Mem/Wb
        - The Rs/Rt register from Id/Ex
- Have to forward from the instruction that is closest to me.
- A load word instruction would mess this all up because the real value is not ready after the Ex stage, it will be ready after the Mem stage. In this situation, we can only forward from Mem/Wb. This means we have to bubble for one cycle, and *then* we can do forwarding. And then you can forward through the transparent latch for any subsequent instruction.
- The case where the LW is followed immediately by a instruction that needs it would need you to stall, you cannot forward in that case. The LW would be in the EX stage and the next instruction would be in ID, but a bubble is the only thing that can be done since the real value isn't available till Mem stage.
    - We check the Memread flag in the Id/Ex latch in order to determine this case.
    - If (ID/EX.MemRead and ((ID/EX.RegisterRt == IF/ID.RegisterRs) or (ID/EX.RegisterRt == IF/ID.RegisterRt))) then we stall for one cycle.

2/21 - Week 7

- If the branch is taken, the the control will go to the target address. If not taken, then we will just go to PC + 4.
- Superscalar means that we have more than one instruction/stage in one cycle.
- Conditionals are like ifs while unconditionals are like goto.
- Direct branch is where address is specified by the instruction. No looking up in register file or anything. Everything you need to figure out the target is in the instruction itself.
    - J and Beq are examples of this
- Indirect branch is where you have to redirect to a RF and grab the address from there.
    - Jr is an example of this.
- Direct branches are easier to predict because they always go to the same place and we just need to be able to predict whether it gets taken or not. Indirect has that plus trying to guess the address itself.
- Unconditional and direct branches are easyyyy.
- Easiest way to handle branches is to just stall until the branch hits the MEM stage, at which point you can fetch the correct instruction.
- When thinking about how to handle these branches, need to consider the penalty (time taken as a result of stalls) as well as the frequency of penalties (what percentage of your instructions are branches).
- Ways to decrease the penalty include trying to put additional logic in the ID and EX stages (a comparator for example), so that we can resolve the branch and set PC accordingly without having to wait till MEM.

    - The downside is that you have complex hardware and you could increase CT by adding to the critical path.
    - Another downside is that our forwarding logic gets messed up since you need to forward the value into the ID stage (to do the comparison), instead of the EX stage.
- If we are guess if the branch gets taken or not, and we're incorrect, we need to flush the the previous instructions and change the PC to the correct one.
    - To flush, we set the control bits to 0 for those instructions.
    - Normally, we kill the bits in the IF/ID latch, the ID/EX latch, and the EX/MEM latch depending on how many instructions we want to remove.
- The easiest guess is a static guess of not taken, since we don't have to care about the target address, because we're going to PC + 4 every time. We also don't even need to know that the operation is a branch.
- Pattern History Tables give you prediction information about each of the branch PC values. The PC value will get hashed, we'll then index into the table, and we'll look at what the corresponding taken/not taken prediction is.
    - Once the branch is resolved, we look at the result and then update the table
- This dynamic approach is a way for you to guess for each branch, without having this one global strategy. We're able to individualize the approach.
- Branch target buffer is like a cache that is indexed by PC, and the BTB tells us the target address for the Taken branch and the type (unconditional or conditional).
    - We would only put branches in the BTB.
    - The purpose of all this is that we can predict that target address all in the IF stage by doing this BTB lookup instead of doing it in the ID stage.
- Branch delay slot says that after every branch, the software leaves a slot after every branch, so that we can insert an independent instruction that doesn't depend on the branch.
    - In a worst case, you would have to put a no-op if there are no independent instructions that fit that criteria.
    - The no-ops would potentially cause an increase in IC, while CPI stays relatively the same.
- Benefit of hardware branch prediction is that you don't have the software having to know about hardware details. In delay slots, they do have to communicate. One drawback with delay slots is that it becomes tough to find those independent instructions sometimes.
- TCPI = BCPI + MCPI
    - BCPI - Baseline CPI takes into account the baseline of processor
        - Function of Peak CPI and data hazards and control hazards
        - Data hazards calculation will depend on the amount of 1 cycle and 2 cycle stalls, as well as the percentage of those occurrences.
    - MPCI - Memory CPI
- Data hazard calculation: Need to determine what the percentage of instructions are A -> D or A -> I -> D or A -> I -> I -> D, and then the corresponding delays. It also depends on whether we're just using stalls or if we're also using forwarding.

    - The main thing is that we're summing up Penalty * Frequency for each situation.
- Control hazard calculation: For the static not taken prediction, and if branches are resolved in memory, and 10% of your instructions are branches, and they are taken 50% of the time, the delay would be (.1)(.5)(3). The 3 because it will take 3 cycles to recover if the branch was incorrect.

2/26 - Week 8

- In the case of an exception, it has to provide a precise state of the hardware, and go to the code that handles it.
- If an exception happens, the instructions following it should not be affected. The state of the processor at the time of the exception is the one that gets sent to the software that should handle it. This is a precise exception.
    - If you have LW, ADD, DIV, SLT, SUB, we want to provide the state of the processor exactly as it did after LW and ADD occurred.
        - Need to provide the state to the exception handler. Need to redirect the PC to that code. It treats the offending exception as a sort of branch to the handling code.
    - This gets messed up with out of order execution.
- Superpipelining means that we want to improve cycle time.
- Pipelining in the beginning originally wanted to break up the execution into different stages and to lower the cycle time. Superpipelining wants to lower it even further.
    - Individual latencies will be worse, but overall will probably be better.
- Growing the pipeline would increase the chances of dealing with hazards. By increasing the pipe depth, the penalty will also increase how many instructions will get removed. The penalty is however many instructions that have gone in before the EX stage. In normal pipelining we'd remove those in IF and ID, but in a 10 stage pipeline, we'd remove those in IF1, IF2, ID1, ID2, and EX1.
    - All of this impacts CPI and pushes further from 1.0. Tradeoff with superpipelining is that we decrease cycle time but CPI could increase because of the penalties from data and control hazards.
- Load word followed by a dependent instruction is the worst case in pipelining, since even with forwarding, you have to stall for once. In the cases where you have more than 5 stages, the delay will likely be even worse, and then you will have to stall for LW -> I -> D.
- Add stages to MEM and EX is what makes things worse for forwarding and load use instructions.
- Superscalar means that we push more than 1 instruction in one stage simultaneously. This would decrease CPI. If we do 2 instructions per cycle, CPI would be 0.5.
- In the IF stage, we would have to take in a PC, and then instead of taking out 32 bits, we take out 64 bits which means two consecutive instructions, but we want those to be independent.
- In the ID stage, the register file would need to double the number of ports.

- We also can't support two loads or two stores or two R types in one stage.
- Compiler can also see what the program is doing so that we can bundle two instructions.
- If we move ADDI up in the pipeline, then we need to make sure that we change the offset for the subsequent instructions that use whatever register is getting updated.
    - We won't be able to move ADD (instead of ADDI) since we cannot decrement by a register value by another register value, but we can decrement a register value by an offset.
- The point of loop unrolling is to unveil more parallelism. You look at multiple copies of the loop. The downside is that you have to have more instructions in memory, but the upside is that because we have more instructions, then we have a greater pool to be able to extract parallelism for.
- First, we loop unroll, then we do some code condensing (normally combining the ADDIs and changing the offsets for stores), and then we have to do register renaming because if it we didn't, we probably wouldn't be able to execute the instructions in parallel such as when you have two operations on the same register. The cost of renaming is that register pressure goes up (more registers, more spills, etc)
- The registers that get renamed will be from the 2nd iteration instructions of the loop.
- Two instructions in the same stage of the pipeline won't see the impact of each other. For example if you had an ADDI and then the SW in the same cycle, the SW wouldn't see the impact of the ADDI on the register value.

2/28 - Week 8

- No way to build a memory that is both large and fast. Therefore we have a hierarchy.
- The closest cache to the CPU is L1, and this will be the first place we look. There may also be a L2 cache that is larger, but slower.
    - L1 -> L2 -> L3 -> Mem -> Disk
    - As you go to the right, latency will increase as will the size
- Instruction memory and data memory are replaced by instruction cache and a data cache. They are part of the pipeline and they contribute to the latency.
- Cache is in a physically different location from memory.
- The reason we have fast and small at the top and large and slow at the bottom is because we can try to exploit locality and we'll end up using the cache more often than having to go all the way to memory.
    - Temporal locality and spatial locality
    - Keep things that you use frequently closer to you so that it will be faster to access
- Parameters for a cache
    - Size: Defined by the data that it holds, not by the bits required to implement it
    - Associativity:
    - Block Size:
    - Replacement Policy:
- Cache itself will get broken up into blocks.

  - Let's say every block in the cache is 16 bytes, and the cache contains 4 blocks. Thus, our total cache size is 64 bytes.
- The block size of the cache is the granularity of the movement, meaning in this case 16 bytes will be taken in at once, in order to amortize the cost of having to go to memory. We don't want to just get one instruction at 512 - 516, we want to get 512 - 528.
- When you have an instruction, it will go to cache, and the cache will either have that instruction or not. If it's not there, we have to go to the next level of the hierarchy and bring in block size amount of data into the cache.
- In your 32 bit cache address, the first 28 bits will determine which block to get from memory (it is called the cache block address), and then the next 4 bits tell you which 1B quantity to address into.
- In addition to storing the 16 bytes in each cache location, we need to include a tag that tells us the block number. That tag will be 28 bytes since there are 2^28 block locations in memory. This is the case when we have a fully associative cache. We also have a valid bit for each cache location to tell us if there is something in the cache location.
- As you go further down, the block size may increase which decreases the number of blocks and thus decreases the number of bits you need in your tag.
- If you have a given instruction, and you miss in the cache, then you add the tag to the first entry of the cache, and you add M[tag] to the data portion of the cache entry, and you set the valid bit to 1.
- From the cache's perspective, it's loading in data at a block size granularity. It's not dealing with that offset. The datapath will use that offset to access values in that 16 byte block.
- Cache needs to have a replacement strategy for when the cache is full and everything is valid. We need to evict something out of the cache. LRU is a common approach. FIFO, random, and MRU are others.
- Associativity is thinking about what blocks can go in which location in the cache. Fully associative means that you can place any block into any location. Direct mapped means that a block can only go to one and only one location in the cache.
  - For fully associative you have to do a linear search for all the tags to determine whether you have a hit or a miss. Direct mapped will only need one comparison because there is only one possible location for the block.
- One of the problems with direct mapping is that it is less flexible in that all N instructions in your working set can map to the same cache location. The rest of the cache may have space, but it won't get used.
- In direct mapped, we need to have a tag, index (for determining which cache location we want to go to), and offset. Now, we would need only 26 bits for the tag. Each of the cache locations can map to 2^28 / 2^2 different blocks.
- Even in direct mapped, we're first check the index, then bring in the tag for the instruction, and then M[tag] for the data portion. If the next instruction maps to the same cache location, then great if it's the same tag but if it's not, then we need to evict it and replace it.

- Set associative means we'll have a bunch of indexes, and at each index, we have a couple of buckets to store data.
- K-way set associative cache means that there are K buckets for each entry in the cache. If the number of ways is 1, then you direct mapped. If the number of ways grows to the number of entries (and thus you'll only have one entry in your cache table and number of buckets will equal number of entries), then it'll get to fully associative.
- If we have 16 entries in our cache, 32B block size, then we have 2^27 total blocks.
  - In FA, we'd have 27 bit tag and 5 bit offset.
  - In DM, we'd have 23 bit tag, 4 bit index (because we have 16 entries), and 5 bit offset
  - In 2-way, then we'd have 8 entries in the cache, with 2 buckets each, and then we'd represent that with 24 bit tag, 3 bit index, and 5 bit offset.
  - In 4-way, then we'd have 4 entries in the cache, with 4 buckets each, and then we'd represent that with 25 bit tag, 2 bit index, and 5 bit offset.
- For k-way instructions, we look at the instruction, look at the index bit(s), go to that entry in the cache, and then look at all of the k buckets to see if it's there. If it's not there, load in the tag and load in M[tag]. Otherwise, great if it is there.
- If you have 32KB cache size, 4 way associative, and 128B block size, then what's T/I/O?
  - It will be 19, 6, 7.
- 32KB = 2^15 because 32B = 2^5, 64KB = 2^16 because 64B = 2^6, etc
- If you have 2MB cache size, 8 way associative, and 256B block size, then what's T/I/O?
  - It will be 14, 10, 8.
- Index = cache size / (block size * bits for n way)
- When dealing with stores, this is a question of if a write modifies something in the cache, then should we update memory or not? The options are to write to memory immediately or just save a bit and then write to memory later. It's a question of when we do it.
  - Depends on how much we deal with cache versus how much we want to go to memory.
- Write through is the option of writing immediately to the next level.
- Write back means that we delay the write until later. Basically, we will make the write when we evict the block.
  - The dirty bit will indicate that the cache block has been written to and thus we need to fix it in the lower places in the memory.
- Pro with write through is that we have consistent state for multithreaded programs. Gives you an immediate update.
- Pro with write back is that if we're making a lot of updates to one particular value, we can just make one final modification to memory at the end instead of doing it after every single update. Don't waste multiple writes when you can instead exploit locality and just do the operation at the end.
- Write allocate and write around deal with cache misses on store words.
- If we have some SW A (write request) and some A is not in my cache, do we bring the block into the cache?

- Write allocate means that we bring that block into the cache and modify it. This helps with exploiting locality because it likely means that we will use it later on.
- Write around means that we don't bring it in, and instead just writes to it in the memory. This helps in cases where we don't want to pollute the cache with this value, which may or may not be used later on.

3/5 - Week 9

- TCPI = BPCI + MPCI
    - BCPI = peak CPI + Data Hazards + Control Hazards
    - MCPI = Instruction Memory + Data Memory
- Normally, in your L1 cache you'll have two sections, one for data cache and one for instruction cache.
- Each cache has a corresponding miss rate as well as the latency associated with it.
- Data cache is within the MEM portion of the pipeline while instruction cache is within the IF stage.
- Let's say you have a 32 bit PC, you break it up into T/I/O for when you look in the L1 cache (specifically the instruction memory cache). If you miss, then you have to go to L2, which will definitely have different characteristics, and thus a different T/I/O.
    - Basically the address can be viewed in different views based on the characteristics of the cache it is going to.
- AMAT is the average memory access time, starting from the top level.
- AMAT of the data cache is time (taken to determine if there is a hit) + miss rate * miss penalty
    - If let's say we have 2 cycles for determining if there is a hit, miss rate of 0.2 for L1 cache, 20 cycles for L2 cache, 0.05 miss rate for L2 cache, and 100 cycles for memory.
    - AMAT = 2 + (.2)(20 + .05(100))
- We do this for both data cache and instruction cache. Then, to get the full AMAT, we use AMAT = (percentage of instructions that are LW or SW)(data cache AMAT) + (1)(instruction cache AMAT)
- BCPI = peak CPI + (% LW)((% LW that are LW->D)(penalty) + (% LW that are LW->I->D)(penalty)) + (% Branch)(Miss rate)(penalty)
    - This isn't concerned with misses or anything, it's just concerned with the delays associated with the impact of having to stall if there is a load word and then a dependent.
    - The LW->I->D situation may or may not be required. In a normal 5 stage pipeline with forwarding, it is normally not required.
    - This BCPI is really really dependent on how many stages your pipeline is made of and if you have forwarding or not.
- MCPI = (1 - B/c every instruction accesses instruction memory)(miss rate)(penalty) with the same process as AMAT + (% instructions that access memory - LW and SW)(miss rate)(penalty)

    - We're basically computing the same thing as AMAT, but we are not considering time taken to determine if there is a hit.
    - MCPI is just about what happens on a miss.
- 3 types of cache misses
    - Compulsory: Miss that you couldn't have avoided with a larger cache because you've never seen that address before. Size and associativity don't matter. It's like a cold start miss.
        - When you start with an empty cache, most of your misses will be compulsory.
    - Conflict: Miss where you had enough space, you had seen it before (but it got kicked out). If you had grown the associativity, then it woulda been chill.
        - It'll be an address that you've already seen
    - Capacity: If in the same situation as above, and you would have missed even with a fully associative cache.
        - Mostly happens with really small caches
        - We basically want to compare the current situation with a model for if you have a fully associative cache, and if you still would have missed, you have a conflict problem. If you would have hit, then you'd have a conflict miss.
- When you have a cache problem where you have a bunch of instructions, first determine the T/I/O values so that you can break up each of the instructions in the right way.

<u>**3/7 - Week 9**</u>

- Each of the applications running thinks that it has all 4 GB of memory to itself.
    - The applications view of its 4GB memory is actually mapped to some physical memory or mapped to disk (in which case, we'll have to deal with the latency of bringing it into memory).
- Plus side of having virtual memory is that compiler doesn't need to know what other applications are running at the same time. Programs A and B can also share virtual addresses (512 for example) because they can point to different locations in physical memory.
- Having this virtual memory means that if we have an instruction LW X, then X refers to a virtual address, and then we have to go to the page table and see what place in physical memory it maps to, and then perform the load word with that address.
- Page is to Block, and Fault is to Miss
- A page fault means that your particular page is not in memory, and thus you have to get it from disk.
- The max number of pages in VM = VM size / Page size
- The max number of pages in PM = PM size / Page size
- Each address (like with caches) is broken up into page tag + page offset.

    - If your page size is 16 KB, then you'll have 14 bits in the page offset. This leaves 18 for the tag, and this indicates how many bits it would take to identify a unique page.
- That page tag is what indexes into the page table, from where we get the physical address mapping. You can also have a page fault if you don't see that particular tag in the page table.
- When we obtain our 19 (in the case of 8 GB physical memory) bits for PM, we also transfer over the 14 bits from the offset to get your 33 bit value.
- The cache of the page table is called a TLB. It is a hardware structure that holds a subset of the page table, which will allow us to exploit locality.
- If there is a TLB hit, then we don't have to go to physical memory, and get the value.
- Main difference with TLB is that we're using the tag portion to break things up.
- Physically Indexed, Physically Tagged (PIPT) means that caches only use physical addresses. Virtual address gets translated to physical address (through TLB) and then we go to the cache.
- If the TLB hits, then we go to the cache, then L2, then Mem. If it does not hit, then we need to go to the actual page table to perform the load and then you check cache, then L2, then Mem.
- Using index to determine which row in the cache, and using the tag for the comparison.
- VIVT means that we split up the whole address into T/I/O and we use those values to look into our cache.
- VIPT is the compromise approach where we only use the virtual address index in the cache.
    - Don't see any additional latency when you're going into cache, as long as the latency involved in going to TLB is similar.

3/9 - Week 9

- In all write operations (store word for example), we can always start the write to the cache and the write to the memory at the same time regardless of whether there is a hit or a miss.

3/12 - Week 10

- With single threaded CPUs, we're trying to exploit Instruction level parallelism.
    - Trying to overlap instructions in time or reorder code, but this is complex to do in hardware.
- In thread level parallelism the compiler breaks the program into multiple threads and now the hardware can exploit its ILP on each thread.
- The number of threads you have may influence how much ILP you need to extract from each thread.
- ILP doesn't have compiler help, but TLP does.
- MLP tries to overlap memory accesses that you know will miss. The idea is that you put them right after one another in order reduce the total time we will have to wait.

- SISD is single instruction stream and single data stream.
    - Doesn’t extract any thread level parallelism
- MIMD is multiple instruction stream and multiple data stream.
    - Happens when you just duplicate the hardware (PC/IF/ID/EX/MEM). You could have multiple threads of execution because you have two copies of the pipeline.
    - Different from superscalar because in that, we want to be able to have more than one instruction in each stage. But here, we just replicate everything.
    - Extracts thread level parallelism. But you need to have a lot of applications running to extract that performance improvement.
    - Also called chip multiprocessor
- With sharing the pipeline instead of partitioning, there could be side effects in that one thread could adversely affect the other. However, if let’s say thread A is working only a little while B is working a lot, then sharing would be good because you have better resource utilization.
- Simultaneous multithreading allows for another thread to also execute at the same time in the same single pipeline (could be superscalar or not)
    - Could give overall improvement in throughput, but single thread performance could suffer. For example, one thread running alone would have a cache hit, but once you add additional threads, that could pollute the cache and that could cause cache miss.
- SIMD is a compromise point where you have a single flow of instructions, and you allow it to operate on multiple pieces of data.
    - Helpful for vector operations where you apply one operation to multiple pieces of data.
    - We’d still need to have multiple execution resources (adders) but we don’t need multiple IFs.
    - You’d only need to fetch and decode one instruction and only go to register file once which reduces power needed
- Trying to extract TLP means that you need to think about how the threads will communicate with each other. If thread A has its own memory, and B has its own as well, the communication model is message passing. We don’t have to worry though about them touching each other’s memory. The other model is where you have a shared memory where there is no message passing, but rather you have to deal with race conditions and synchronization primitives.
    - Need to think about the overhead of the communication (separate memory) vs the race conditions you will encounter (shared memory)
- Two cores/CPUs can have their own caches, or they could have a shared cache.
- Snoopy coherence is where you will pay attention to a shared communication resource to detect any changes and perform any invalidations.
    - Invalidating is one approach but you can try to bring in that modified value into your cache which is snarfing.
    - Scaling up snooping is tough because you need to keep listening on every core/CPU

- Directory coherence adds a level of indirection to make this approach more scalable. It works by tracking different blocks and just says whether some value is invalid, valid, or modified, so if another core wants to access the value, it will check the directory and it will see that it is modified. And then the directory can tell the other cache to write through. The cores have to ask the directory instead of it just listening in on the bus. Directory basically tracks the state of individual blocks.
- Directory is better than snooping in that you don't have to always be listening (you'd be wasting a port and power) and you don't have to have that bottleneck, it scales a lot better. The drawback is that it would probably be slower for another CPU to have to ask the directory, it will tell you it's modified, and then the other CPU has to write it, etc etc.
- Direct memory access is like another hardware structure that takes away responsibility from the CPU when it comes to memory access. CPU can focus on doing its own stuff, while DMA engine handles memory.
- An ASIC is a design for a particular application while a CPU is more general purpose. It's more specialized and you don't really need to focus on extraneous stuff.
    - FPGAs are kind of in between and provide extra logic.
- An ASIC is an actual piece of hardware while FPGAs are more flexible in that you can change how they are programmed. Cannot really change an ASIC once it is created since it's actual hardware that is baked in.

3/14 - Week 10

- Overview of the class
    - Performance metrics
        - Equation for ET. ET = IC x CPI x CT
    - ISA
    - ALU
    - Single cycle datapath
    - Pipelined datapath
        - Data (software - reordering and noops, and hardware - stalling (hazard detection) and forwarding (forwarding logic) solutions) hazards
        - Control hazards: Software - branch delay slots. Hardware - static and dynamic branch prediction
        - structural hazards
    - Superscalar - more than one instruction per cycle and thus CPI < 1.0 . Trying to reduce CPI
        - There is a scheduling problem where you're trying to figure out how to package multiple instructions together. You can do this through software through loop unrolling which uncovers more parallelism.
        - In terms of hardware, you have out of order and dynamic scheduling.
    - Superpipelining - Trying to reduce cycle time by increasing the number of stages in your pipeline.

- TCPI = BCPI + MCPI, and we said BCPI = peak CPI + Data Haz + Control Haz and MCPI is just for cache misses
    - Calculating these values always involves frequency and penalty of the hazard.
    - BCPI takes into account the hit path while MCPI is about the misses and the impact in the caches
  - Hit path with the TLB
  - SMT, CMT, Cache Coherence SIMD, DMA
- Important part that with forwarding, you're passing from the end of EX (end of M in the case of load word) in one instruction to the beginning of the EX in the next instruction.
- If a branch is resolved in EX, then the correct instruction (if you predicted wrong) would come in in the 4th cycle.
- For branch delay slots, if the branches are resolved in EX, then the compiler would like to add in 2 independent instructions or 2 no-ops after the BEQ.
  - When choosing what instruction to place into the slot, you need to look at what registers are being used in both the taken and not taken paths. If the same register is used in both paths, then you can put only one of them in the two slots because the register will get modified anyway.
- Software fixes for hazards normally increase IC but CPI still stays close to 1. Hardware fixes normally include stalls which increase CPI but these fixes will almost never affect IC.
- TCPI Example: LW 30 percent, SW 10 percent, Branch 20, R type 40. LW -> D 20, LW -> I -> D 10, LW -> I -> I -> D 15 and you have forwarding and a 6 stage pipeline (2 M stages). Predict NT always, 60% of the time Taken, Branch resolved in EX.
  - BCPI = 1.0 + (0.3)(2)(0.2) + (0.3)(1)(0.1) + (0.2)(0.6)(2)
    - First term is peak CPI, next 2 are for data hazard, and final is for control hazard
    - We pay a 2 cycle penalty in that first scenario (LW -> D) because we are ready to forward after M2, there will be a 2 cycle stall before it can get forwarded to the beginning of EX for the next instruction
    - The last term has 2 cycle penalty because 2 cycle difference between IF and EX.
  - MCPI = (1)(0.1)(25 + (0.1)(300)) + (0.4)(0.25)(25 + (0.1)(300))
    - First term is about missing in the different levels of the cache and the impact of those misses.
    - Second term is for the loads and stores. Review what goes into that. I think it's related to the first term
- For VM stuff,
  - First, find the number of physical pages and the number of virtual pages. Physical bit refers to the offset and the virtual bits refer to a virtual page number
  - Review this lecture video
  - TLB includes the physical page number. Index the tlb with the virtual to get the physical.

- Remember that for FA, you only have a tag and an offset. For DM and k-way, you have a tag, index, and offset. Make sure you calculate that offset.
- TLB page line equals the number of bits in the PPN. Those bits are the page offsets. The more significant bits indicate the virtual page numbers. The bit(s) more significant than that page line is what determines hits and misses.
    - VPN then PPN makes up the TLB. The VPN is made up of a Tag and an index. For 16 translations and 8 way SA, we'd have 1 index bit.

3/16 - Week 10

- TLB is used as a cache for the page table.
- To access the data cache, we can either use the physical address or the virtual address.
    - Normally, we use the physical address.
- What is the problem with using the virtual address for both index and tag?
    - There could be conflicts since virtual address space could intersect between different processes. Different programs might use the same virtual address and map to different physical addresses and we cannot make any distinctions without more bits to identify a particular process.
- One solution to the above is to use the physical address for both.
    - Translate VA to PA through the TLB if possible.
    - Extract the index and tag bits from the physical address and use those to check if the data that we want is in the cache.
    - Use the virtual address for the index into the cache, and physical address for the tag. This gives us parallelism because we can use VA to index into the cache while we can also start translation process to turn physical to virtual address to get our tag so that we can check it in the TLB.
        - First index into the cache using virtual address, and then do the tag comparison later.
        - Aka translate VA to PA via TLB and index cache with VA in parallel. Then, extract tag from PA and compare with cache entry.
- When we're trying to parallelize work across a number of processors, we need to take into account communication overhead, memory considerations, and whether or not we get to assume uniform load.
- In multiprocessors, we could either have one shared physical address space or one private physical address space per processor.
    - With shared physical space, it's easier for one processor to see and operate on the results of another. Harder to implement because we need more hardware logic to make sure that we don't run into memory access problems.
    - With private address space, it doesn't need to worry about stepping one another processor's memory. Honestly, it's much easier to implement too because all we need to do is just message pass.
- Grid computing refers to loosely coupled clusters but over long haul networks.

Video Notes

1_1 - Computer Abstractions and Technology

- Moore's Law is the doubling of transistor technology every couple years.
    - Adding more area gives you more compute.
- Personal computers are more general purpose while server computers are network based and need to be high capacity, performance, and reliability. Supercomputers are made for high end engineering calculations. Embedded components are hidden as components of systems and they often have stringent power and performance costs.
- In the PostPC era, you have personal mobile devices and cloud computing.
- Levels of Program Code: High level language like C, Java, etc -> Assembly language which is the textual representation of instructions -> Hardware representation which are the binary bits which have to be used to actually execute instructions.
    - Plus side of a high level language is that it's portable. You can use those languages to describe your tasks even if you have totally different processors, operating systems, etc.
    - Compiler is the one that translates that high level language down into assembly language.

1_2 - Computer Abstractions and Technology

- The control is the part of the machine that organizes how the information flows through the datapath, which are the channels that handle the processing of the data.
- System on Chip (SoC) refers to the concept that combines the CPU, GPU, processor, etc onto a single piece of silicon, allowing more efficient communication.
- The CPU consists of
    - Datapath: Performs operations on data. Flexible in terms of the operations that it can perform.
    - Control: Sequences datapath and memory
    - Cache memory: Small and fast access to data and it is close to the processor.
- Instruction set architecture (ISA) is the hardware/software interface. It's the language the compiler will use to speak to the processor. It's basically an agreement between the software and the architecture so that they know how to properly communicate with each other.
- DRAM capacity over the last decade has been increasing, but DRAM speed hasn't been, so there's been a gap between processor speed and memory speed.
    - So basically we can support big data in terms of its capacity, but the speed right now isn't there. There is high latency, and thus the solutions need to be parallelism, caching, etc.
    - This is called the memory wall problem.

1_3 - Computer Abstractions and Technology

- There are different ways to measure performance.

    - Response time: How long it takes to do a task
    - Throughput: Total work done per unit time
        - Can think of this as bandwidth because it’s the max amount of work that can get done in a certain amount of time.
    - In some cases, throughput is more important than response time when you care about how much work should be done rather than if one piece of work is done quickly.
- We can define performance = 1 / Execution Time
    - “As the execution time shrinks, the performance increases”
- Measuring execution time
    - Elapsed time: Total response time that includes the processing, I/O, overhead and idle time.
    - CPU time: Time spent just processing a given job. Composed of both user CPU time (time for the actual program you’re considering) and system CPU time (time that the kernel OS takes in servicing the program).
        - The larger percentage of the elapsed time is CPU time, the larger impact the speed of a processor has. More time is going to be spent there.
- Many circuit paths in a processor, each with different delays, and take different amounts of time to grab a signal.
- As a way of stabilizing each of the paths and making sure that the values are ready, a CPU clock is used to regularize the processing that occurs.
- Clock period: The duration of a clock cycle.
    - Normally this is associated with the time taken to complete some data transfer.
- Clock frequency: Number of cycles per second
- Clock period and frequency are inverses of one another.
- CPU time is a function of the number of ticks that occurred in the CPU clock as well as how long each one of those cycles/ticks took.
    - CPU Time = CPU Clock Cycles * Clock Cycle Time
    - CPU Clock Cycles = CPU Time * Clock Rate
        - Clock Rate = Clock Cycles / CPU Time
- Performance gets improved by reducing the number of cycles, reducing the time taken or aka increasing the clock rate. Always a tradeoff between clock rate and clock count.
- The calculation for CPU clock cycles can get broken down even more.
    - Clock Cycles = Instruction Count * Cycles per Instruction (CPI)
    - CPU Time = Instruction Count * Cycles per Instruction (CPI) * Cycle Time
    - CPU Time = Instruction Count * CPI / Clock Rate
- Instruction count is the number of instructions executed over the course of the whole program.
    - Static instruction count: Number of instructions when the program is stored on disk/memory.
    - Dynamic count is more of the actual number of instructions that occur when a program actually gets run.
        - Determined by the program, iSA, and compiler.

- Depending on the complexity of each instruction, they may take different amounts of time. The CPI is the average number of cycles per instruction.
  - CPU hardware has an impact on how many cycles each instruction will take.
  - CPI is also dependent on the particular program and the particular computer.
  - CPI is computed through a weighted average of the cost of the particular instruction * the percentage that it shows up in the program aka relative frequency.
- Algorithm affects the IC, and possibly the CPI.
- Programming language affects IC and CPI.
- Compiler affects IC and CPI.
- Instruction set architecture affects IC and CPI and Cycle time.

1_4 - Computer Abstractions and Technology

- Power has become extremely important, in addition to performance.
- Power considerations led to multicore instead of just putting everything on a single core.
- Power = Capacitive load * Voltage^2 * Frequency
  - You can try to scale the latter two if you're trying to figure out how to reduce the amount of power used.
- The power wall is not being able to reduce voltage further and not being able to remove heat since the components on the chip are so close together.
  - One way to get over that was new advancements in instruction level parallelism which was the hardware basically altering instructions and sequences of instructions so that we can utilize the processor more effectively.
- Multicore processor is where you have more than one processor per chip.
  - Requires explicitly parallel programming. The programmer has to find that parallelism and split the program into different threads instead of the instruction level parallelism where you have the hardware itself executing multiple instructions at once based on its knowledge of the program.
- SPEC CPU Benchmark is a common program used to measure performance.
  - They are a set of programs geared toward testing the CPU performance, and thus is low on I/O, not lot of use of GPU, etc. It's just trying to push the CPU as much as possible.
- Amdahl's Law states that if you're trying to improve some aspect of the computer, then you'll see a proportional improvement in the overall performance.
  - "Choosing to optimize something that accounts for 90% of the execution time will be really helpful instead of optimizing something that accounts for 5% of the ET."

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvemen t factor}} + T_{\text{unaffected}}$$

- MIPS (Millions of Instructions Per Second) is not great as a performance metric because it does not account for the differences in ISAs and the differences in the complexity between instructions.

- ISA is below operating systems and compilers but above the circuit design.

2_1 - Instructions: Language of the Compiler

- Instruction set is the arsenal of instructions of a computer. Different computers can have different instruction sets.
- Cannot run a program compiled for ARM on x86 (different instruction sets).
- The complexity of the instruction set impacts the design choices you make for the hardware as well as the code you write for the software.
- When coming up with a new ISA, the designer has to determine what operations are supported by that ISA.
    - How many operations, which kinds of operations, length (number of bits needed to represent that operation) of operations?
        - Number of bits comes into play when we talk about decoding the operations and fitting in memory.
    - How many operands, types of operands?
    - Size and format of instructions? Just like data, the instructions have to fit in memory as well.
- 2 Main ISA Classes: CISC and RISC
    - Complex Instruction Set Computers - Large number of instructions with many specialized complex instructions.
        - Lots of instructions at your disposal, but you have to make sure that your hardware can support all of them. Software side easy, hardware side hard. Less impact on memory also, because you didn't need as many instructions since you had really complex ones that did the job for you.
        - Can also do translations into micro operations that help to get a lit parallelism to help performance.
        - X86 is a complex instruction set.
    - Reduced Instruction Set Computers - Fewer instructions that the ISA contained, which enabled pipelining and parallelism.
        - Because you have smaller tasks that can be broken down, that allows you to overlap certain tasks and get parallelism and thus performance improvements.
        - MIPS is a reduced instruction set.
- Different classes of operations within an ISA.
    - Arithmetic Operations with two sources and one destination and all arithmetic operations follow this form.
        - Add a, b, c // a = b + c

      The regularity that you get from the common formatting is good because it helps the hardware's job of decoding the information. It's make the hardware architecture less complex. Thus, simplicity favors regularity. Simplicity then gets you better performance at a lower cost.
        - You can place the operands to these arithmetic operations in registers, which are fast hardware memory located on chip. MIPS has a register file

that tells you what values the registers have. It’s much faster to access this file than to access the actual memory location. Compilers handle movement to and from the register file. You can access the register file using 5 bits (to represent the 32 registers that you have) and then once you choose one, then the value is a 32 bit value.

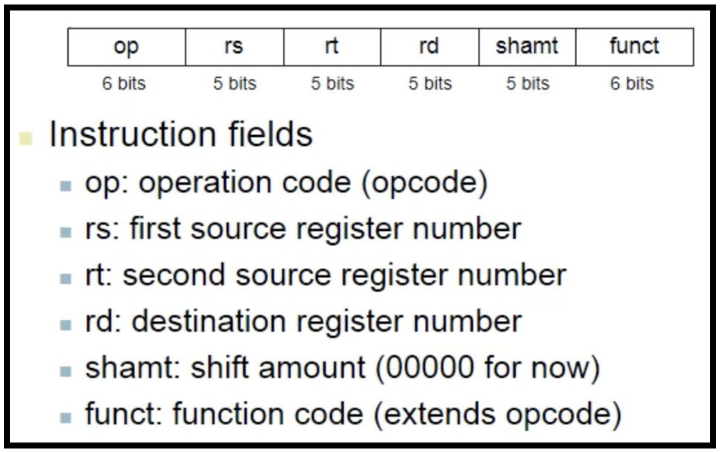
- Having smaller memory is faster because your access latency on those wires is much shorter. Basically related to physical design.
- In RISC, we have explicit instructions to move data from memory into registers, perform some operations on those register values, and then move the data back.
- We can think of memory right now as just being a huge array where each location is identified by 8 bits.
- MIPS is Big Endian, which means the most significant byte is at the lowest address of the word.
- Load word writes a value to a register from memory. Store word takes the value thats in the register and writes that to memory.
- Registers are faster to access than memory. Operating on memory data requires loads and stores into registers.
- Compilers must use registers for variables as much as possible and they should avoid having to spill information to memory if there aren’t enough registers. This spilling will require a lot more loads and stores.

2_2 - Instructions: Language of the Compiler

- Memory and registers operands are two types of operands to operations but another type is immediate operands.
    - They are constant data and they don’t change regardless of input or whatever since they are encoded into the instruction itself.
    - The fact that constants are often small means that we can avoid a load instruction. We don’t need to store that value into a register.
- One example of having a simpler ISA is that instead of having a move operation, you can simply have an add operation where the second operand is a register that always has zero in it. This allows you move values between registers without necessarily having an instruction that explicitly does that.
- Given an n-bit number, the unsigned binary integer representation will have a range of 0 to 2^n - 1.
    - 32 bits gives you a range up until 4,294,967,295
- If you want to represent that same number in 2s Complement, then you can have a range of -2^(n - 1) to 2^(n - 1) - 1
    - For 32 bits, the range is -2,147,483,648 to 2,147,483,647
    - Bit 31 is a sign bit with 1 is negative number and 0 means a positive number.
- If you want to negate a number that is in 2’s Complement form, then you want to complement everything and add a 1.
- To represent a number using more bits (for example, turn an 8 bit number into a 16 bit one), then take the most significant bit (the sign bit) and replicate it to the left.

2_3 - Instructions: Language of the Compiler

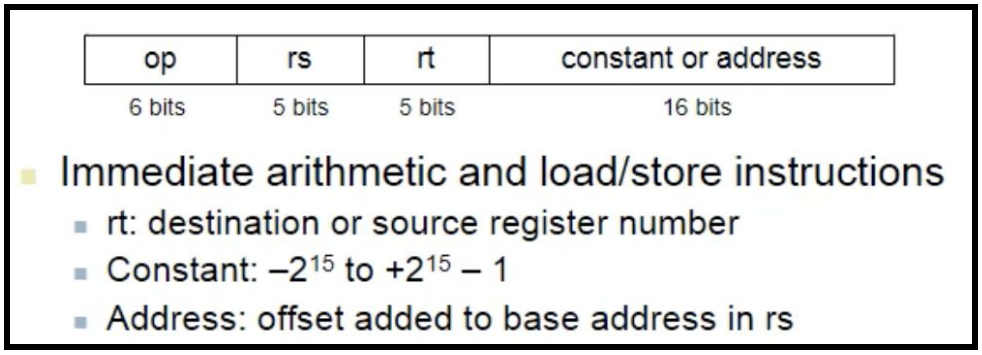
- In MIPS, the instructions in our ISA have to be encoded in binary, and that's called machine code and they're stored in memory.
- Every instruction in MIPS takes 32 bits. This requirement means it's a fixed length instruction set.
- Instructions in MIPS are the same size, the same format, and treat their data in similar ways. This is a key point of RISC architectures.
- One type of instruction is called R-format Instruction.



op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

  - The op code tells us this is an R-format instruction and the function code tells us in particular which R-format instruction/function (add, subtract, etc) we are referring to.
- Another type is called I-format Instruction, which are immediate arithmetic and load/store instructions.



op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

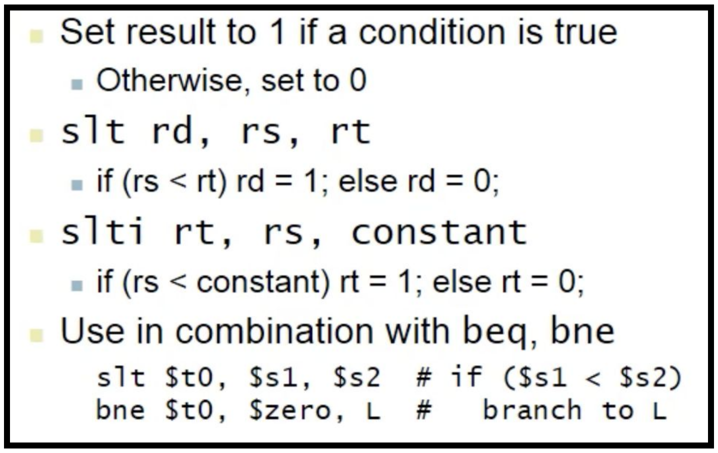
- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

  - Rt is the destination or source register number depending on the instruction and how it is used.
  - One tradeoff is that we don't allow our immediate constant value or address to be more than 16 bits long, because otherwise we wouldn't be able to store the rest of the information needed for the instruction.
    - But I think we take that 16 bit value and then sign extend to make it 32 bits.

- Bitwise manipulation refers to when you don't look at a number as a whole, but rather you perform operations on the individual bits that make up that number.
    - Logical operations: Shift left, shift right, bitwise AND, bitwise OR, bitwise NOT.
- Shift operations are R-format instructions.

2_4 - Instructions: Language of the Compiler

- Conditional operations are where you branch to a labeled instruction if a condition is true.
    - This would result from if you have a while loop or for loop in your software code.
- Beq checks if rs and rt are equal, bne checks if they're not equal, and j just jumps to the specified instruction.
- Because instructions are 32 bit, then let's say the first instruction is at address 500, then the 2nd instruction will be at address 504.
- A basic block is a sequence of instructions where you don't have any embedded branches and you don't have any branch targets, which means no places where a branch can take you into that position.
    - You have a guarantee that your entry will be at the top of the block and your exit will be at the end of the block.
    - The compiler identifies these blocks for optimization.
    - We can also describe the control flow between basic blocks.
- Another operation is slt (set less than)



- Set result to 1 if a condition is true
    - Otherwise, set to 0
- `slt rd, rs, rt`
    - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
    - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

    - Important note is that it doesn't result in a branch but rather in a comparison that changes an output value.
    - We don't have a blt (branch less than) because that would be slower and would mean more work per instruction and thus requires a slower clock. Therefore, it's just better to have slt and then a separate branch instruction that branches based on the output value.

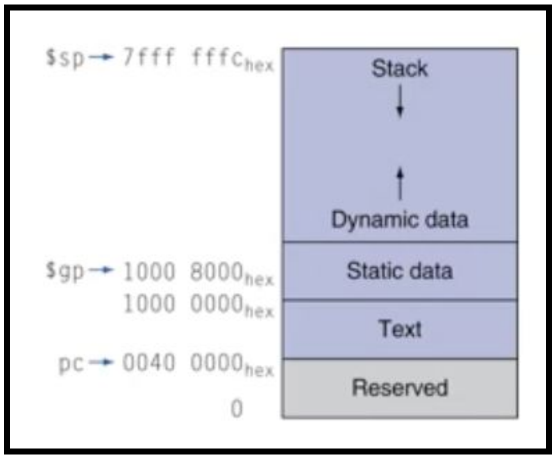
2_5 - Instructions: Language of the Compiler

- Procedure calls are unconditional because they will always happen regardless of a condition. Something else they do is save the PC so that you can return to wherever you

were in the program before you jumped to another procedure. The steps for a procedure call are
    - Place parameters in registers
    - Transfer control to procedure (Do this by changing the program counter)
    - Acquire storage for procedure (Adding to the stack frame)
    - Perform procedure's operations
    - Place result in register for caller
    - Return to place of call and execute next instruction after procedure call
- The $ra register is a special register just for holding the return address from a procedure call.
- The procedure call is composed of a jump/link instruction called jal. It will put the address of the following instruction in $ra so that it can come back to it and performing the procedure and it will then jump to the target address to start that procedure.
    - jal ProcedureLabel will be the instruction.
    - It implicitly uses $ra.
- The procedure return is called jump register and it basically copies whatever is in $ra (so the place that we want to return to) and sets the program counter equal to that.
    - jr $ra will be the instruction.
- Overhead of procedure calls comes from branching and having to add/remove stuff from the stack.
- Non-leaf procedures are procedures that call other procedures, while leaf procedures don't do that.
    - For the nested calls, the caller needs to save its return address on the stack as well as any arguments and temporaries (such as callee saved registers) needed after the call.

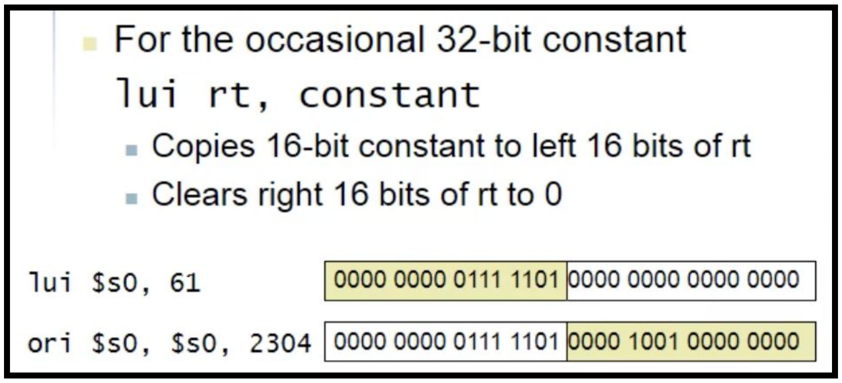
2_6 - Instructions: Language of the Compiler

- Memory layout



    - Text is the program code, static data are the global variables, the dynamic data is the heap (malloc in C) and the stack is automatic storage (for local variables I guess).

    - Program counter will always be pointing to something in the text segment range since that is where all the instructions are held. Instructions themselves are in memory.
- If you have a 32 bit constant, then you can do the following to convert to it from a 16 bit one.

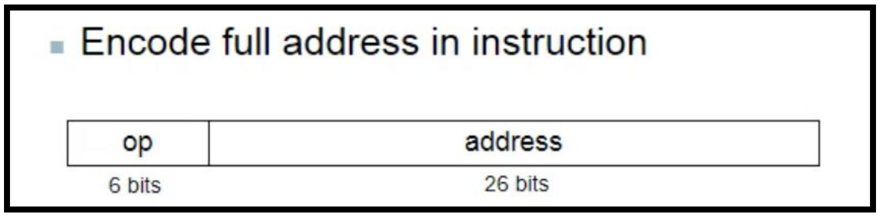


- For the occasional 32-bit constant

  `lui rt, constant`
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

```
lui $s0, 61              0000 0000 0111 1101 0000 0000 0000 0000
ori $s0, $s0, 2304       0000 0000 0111 1101 0000 1001 0000 0000
```

- PC relative addressing is target address = PC + offset * 4. The reasoning is that most branch targets are near the branch so we can use offsets instead of having to specify the full 32 bit location which wouldn't even fit into an I-format instruction.
- J-type instruction has a 6 bit opcode, but then a 26 bit immediate field. These instructions could be j or jal.

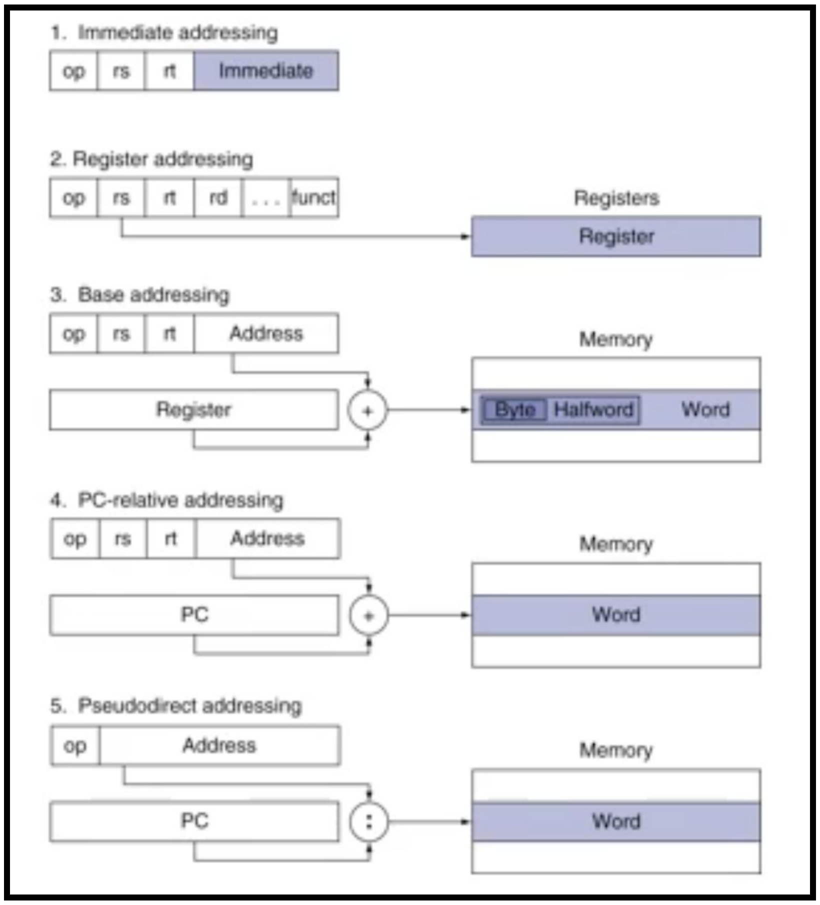


- Encode full address in instruction

op	address
6 bits	26 bits

- Pseudo direct jump addressing is when since you only have 28 bits to describe your address, then you copy over the most significant 4 bits from the PC since you know that where you're jumping to will have those same 4 bits.
- If the branch target is too far to encode with a 16-bit offset, then the assembler will rewrite the code so that you have two jumps, where the first jump will just take you

somewhere closer to your end target.

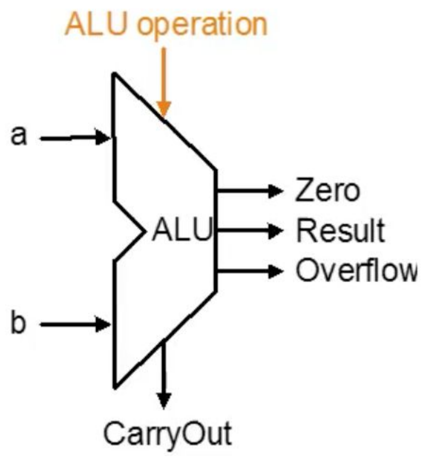


- Addressing Summary

3_1 - Arithmetic for Computers

- Arithmetic for computers involves operations on integers (add, subtract, multiply, etc) as well as dealing with floating point real numbers (the representation of them and operations with them).
- With integer addition, we get overflow if the result is out of range. When adding a positive and negative number, you never get overflow. When adding two positive numbers, you get overflow is the result sign is 1. When adding two negative numbers, you get overflow is the result sign is 0.
    - You can think of addition as being made up of operations where you're taking in 3 numbers (2 bit numbers and 1 carry bit) and you're outputting the value plus a carry bit.
- Integer subtraction involves the addition of the negation of the second operand. When subtracting two negative or two positive numbers, you'll never get overflow. When subtracting a positive from a negative, you get overflow if the result sign is 0. When subtracting a negative from a positive, you get overflow if the result sign is 1.
- When dealing with overflow, some languages ignore it, some languages require raising an exception.

- Inside of an ALU, the process for executing an instruction is that first we fetch the instruction, then instruction decode, then operand fetch, then execute using the ALU, then store the result, and then get the next instruction.
  - Main thing to remember is that the ALU is not the only piece of hardware that is getting used when we want to do some sort of arithmetic computation.



- Composition of an ALU
- Result will always be the same length as the input operands a and b (At least for the addition and subtraction commands).
- The multi-bit ALU is one that is composed of a bunch of one bit ALUs.
- Every time you use the one-bit ALU, the AND, the OR, and the adder components are always used regardless of what the actual operation was that was passed in.
- Difference between half adder and a full adder is that with the full, you have 3 inputs (a, b, and the carryIn), while the half adder has 2 inputs.
  - The full adder outputs the sum and the carryOut.
- The computation for coming up with the carryOut value can be made into a sum of products form that is shown below.

• CarryOut = (!a & b & CarryIn) | (a & !b & CarryIn)
  | (a & b & !CarryIn) | (a & b & CarryIn)

• CarryOut = (b & CarryIn) | (a & CarryIn) | (a & b)

- The sum logic is shown below

• Sum = (!a & !b & CarryIn) | (!a & b & !CarryIn)
  | (a & !b & !CarryIn) | (a & b & CarryIn)

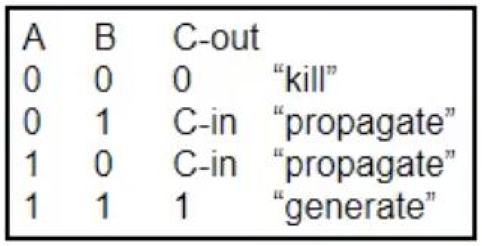
3_2 - Arithmetic for Computers

- In a 32 bit ALU, you'll have 32 separate 1-bit ALUs which do the computation for each of the spaces in the 32 bit integer.

- Carry out of the previous ALU is routed to the carry in of the next ALU.
  - This architecture is called the ripple carry ALU.
- To expand our 1 bit ALUs, we need to include an inverter in our ALU so that we can take the 2's complement of the 2nd operand.
  - We'll have an inverter, but then we also have to add 1 (since that's the procedure for two's complement) so we make the carry in of the first ALU one whenever we want to subtract.
- For an N-bit ALU, we detect overflow by doing CarryIn[N-1] XOR CarryOut[N-1]
- Zero detection is done by just determining if all the results are 0, and so we have one big NOR gate with a fan in of 31. Any non-zero result will immediately cause the output to be 0.
  - FYI that a NOR can be done when you invert both the operands and then AND them together.
- Set-On-Less-Than is an R type instruction that does not change the PC, but only writes to the register file. Basically it is an instruction that compares rs and rt, and returns a 1 if rs < rt, and a 0 otherwise. The way the comparison is done in the ALU is by computing rs - rt.
  - Output will always be 31 zeros and then a zero or a one.

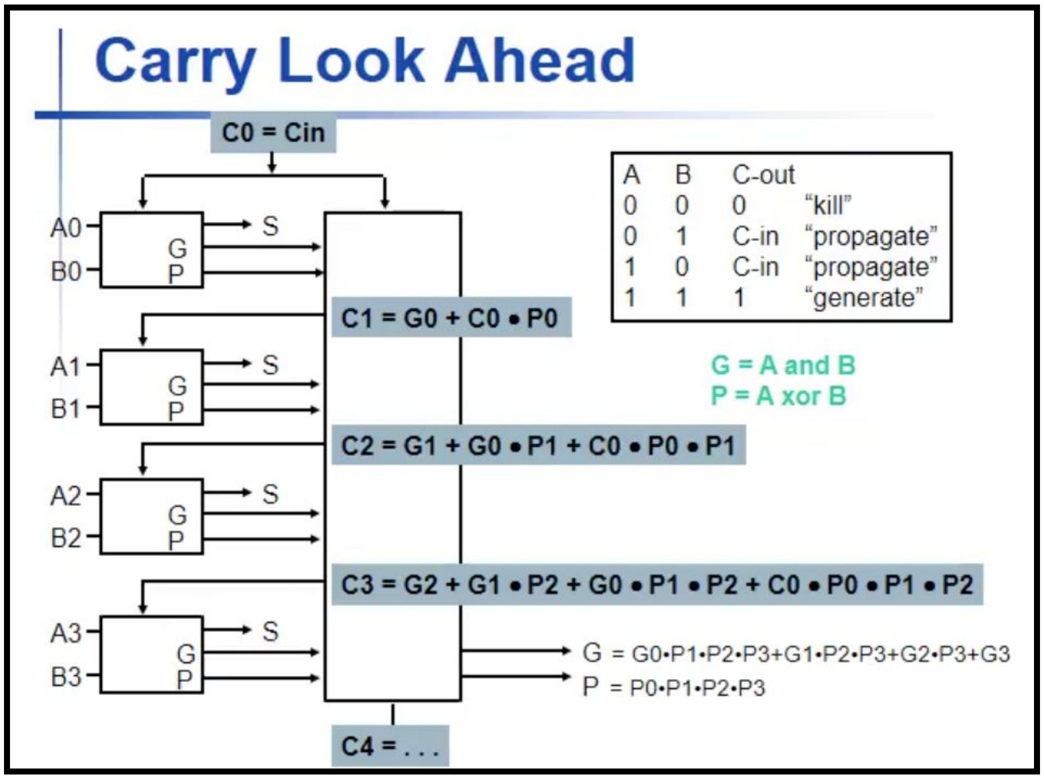
3_3 - Arithmetic for Computers

- The adder has the longest latency delay in the ALU.
- The carry in input is the one that causes some latency since it has to be computed by previous components and thus the unit may have to wait for its value.
- For one bit of computation, there are 2 gate delays per carry out. For N-bit ripple carry adders, we'd have 2N gate delays.
- To fix this, we can try to compute the carry in ahead of time with some additional logic. We want to compute it faster than the time it takes to ripple through all of the components.
- Carry Look Ahead uses the information we know about a and b to predict whether a component will require a carry out or not.



A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

- Table that shows the possibilities.



Carry Look Ahead

A	B	C-out	
0	0	0	“kill”
0	1	C-in	“propagate”
1	0	C-in	“propagate”
1	1	1	“generate”

G = A and B
P = A xor B

$C0 = Cin$

$C1 = G0 + C0 \bullet P0$

$C2 = G1 + G0 \bullet P1 + C0 \bullet P0 \bullet P1$

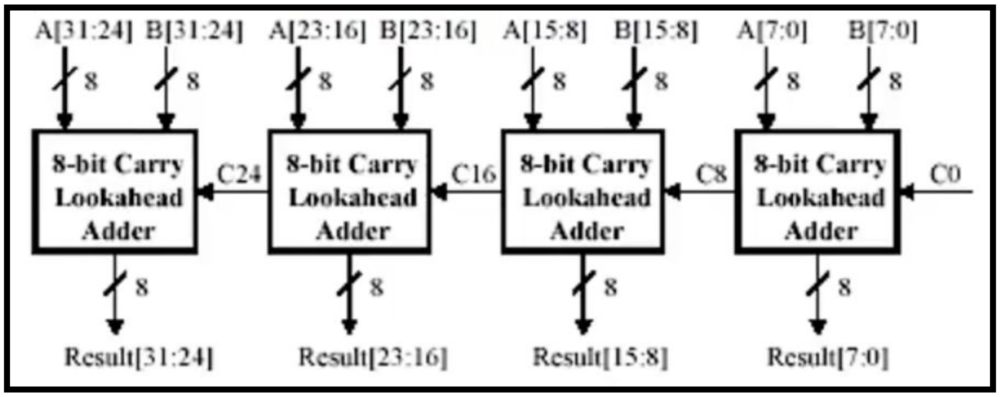
$C3 = G2 + G1 \bullet P2 + G0 \bullet P1 \bullet P2 + C0 \bullet P0 \bullet P1 \bullet P2$

$G = G0 \bullet P1 \bullet P2 \bullet P3 + G1 \bullet P2 \bullet P3 + G2 \bullet P3 + G3$

$P = P0 \bullet P1 \bullet P2 \bullet P3$

C4 = . . .

- The carry look ahead diagram above shows how we can compute values like C1, C2, and C3 all without even connecting it to the carry out of the previous unit. We can do all of those computations in parallel. We simply just compute at as it only depends on the values for a and b.
  - Using a CLA is a tradeoff with space and with speed.
- All the G and P computations get done in parallel since they don’t depend on each other at all.
- All of these extra computation is making sure that you get faster access to that carry in.
- The partial carry lookahead adder is where you connect several N bit lookahead adders together. 4 8-bit carry lookahead adders can form a 32-bit partial carry lookahead adder.



- Important to note above that C8 will act as carry in for the next carry lookahead adder.

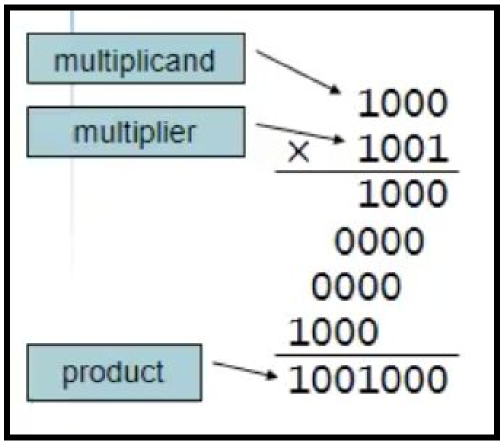
- You can also have a hierarchical CLA where you have 4 4-bit CLA where each of them computes a separate G0 and P0 for that unit. These values are functions of what the individual G and P values are for each of the 1 bit adders.

$$G_\alpha = G0 \cdot P1 \cdot P2 \cdot P3 + G1 \cdot P2 \cdot P3 + G2 \cdot P3 + G3$$
$$P_\alpha = P0 \cdot P1 \cdot P2 \cdot P3$$

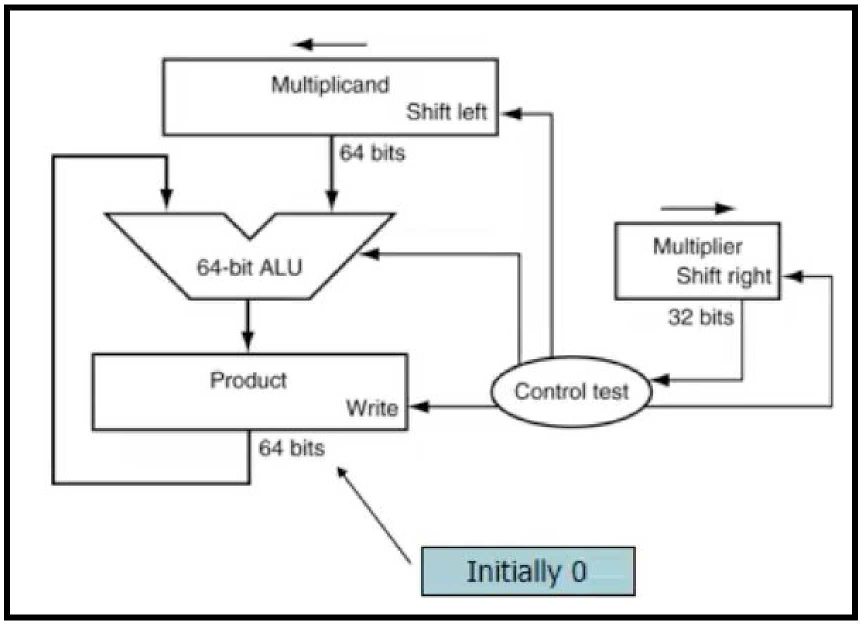
- Generate and propagate works by saying that if you generate at one bit location, then you have to propagate at every proceeding location as well.
- In a hierarchical CLA, all the individual generates and propagates for the one bit adders would be computed in parallel, and then the G0 and P0 for each of the 4-bit adder as a whole will be computed in parallel.
- Carry select adder trades even more area for performance. It is composed of 2 1-bit adders where one if fixed with a carry in of 1, and one has a carry in of 0. Then, the inputs x and y are fed into both ALUs, and then we basically use a multiplexer to decide which one we want to truly return after computing both values.
    - If the delay of the multiplexer is less than the delay from the chained series of ALUs, then it would be very beneficial to do this.

3_4 - Arithmetic for Computers

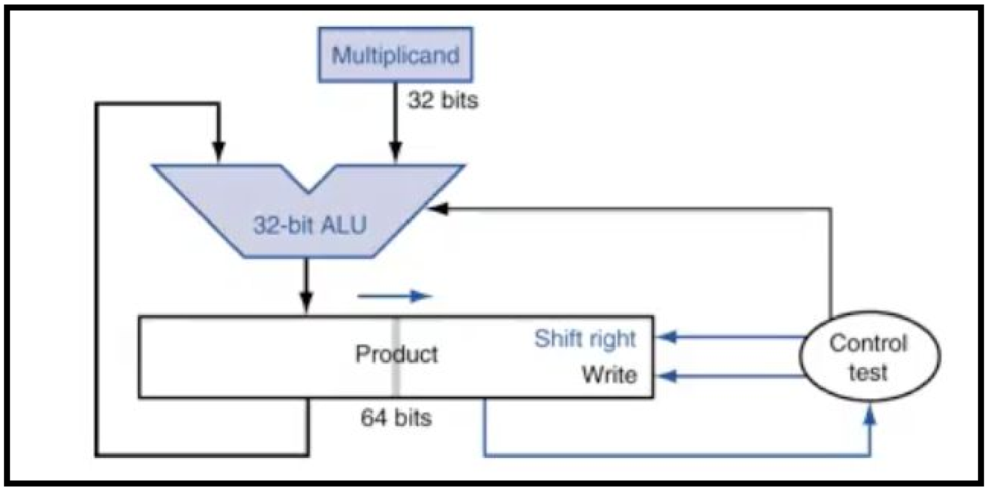
- In binary multiplication, the length of the product is the sum of the operand lengths.
    - Basically, the multiplicand is written out every time there is a one in the multiplier (starting from the least significant bit), and shifted to the left otherwise, and then there is a shift every time. And then the numbers are added together.



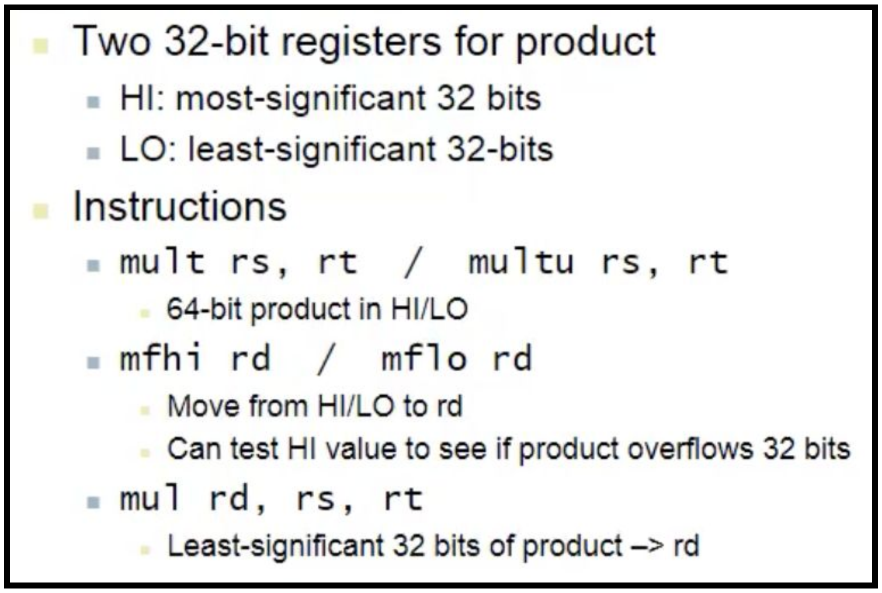
- Multiplication example - Basically a bunch of shifts and adds.
- Multiplication with N bit numbers requires N intermediate values to be added together.



- Even though we're multiplying 32 bit numbers, we need a 64 bit ALU (since we are doing shifts) and a 64 bit register to hold the product value.



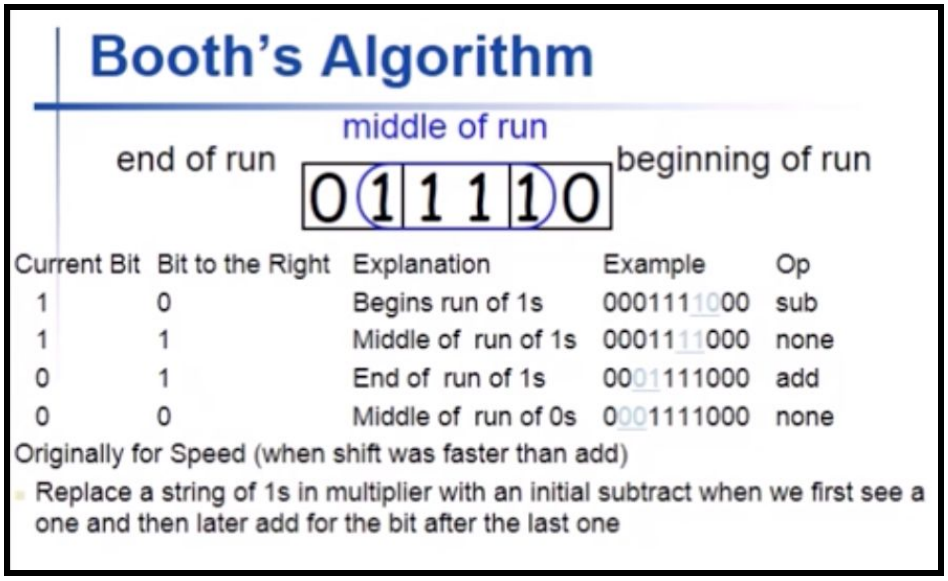
- To optimize performance, we can try to use a 32-bit ALU (since technically we only have 32 bits available at a time).
    - Instead of shifting multiplicand to the left, we add to the upper 32 bits of the product and we shift right. They start up higher in the product, but we eventually shift them lower.
    - Also, the multiplier gets loaded into the product initially, instead of having the product as just 0 to start off with.


- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

- MIPS multiplication above

3_5 - Arithmetic for Computers

- To do signed multiplication, you can choose to make both operands positive, and then just remember to complement the product when you're done by applying 2's complement.
- **Booth's algo is an elegant way to multiply two signed numbers.**
  - The idea is that we want to replace a sequence of additions with a subtraction and an addition.
- If you have a sequence of numbers x1 + x2 + x3, the idea is that you can left shift the highest number by 1, and then subtract the lowest number, and you'd get the equivalent of all of those additions.
  - Reduced a sequence of adds to one add and one subtract.


Booth's Algorithm

middle of run
end of run
beginning of run
0 1 1 1 1 0

Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Begins run of 1s	0001111000	sub
1	1	Middle of run of 1s	0001111000	none
0	1	End of run of 1s	0001111000	add
0	0	Middle of run of 0s	0001111000	none

Originally for Speed (when shift was faster than add)
- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

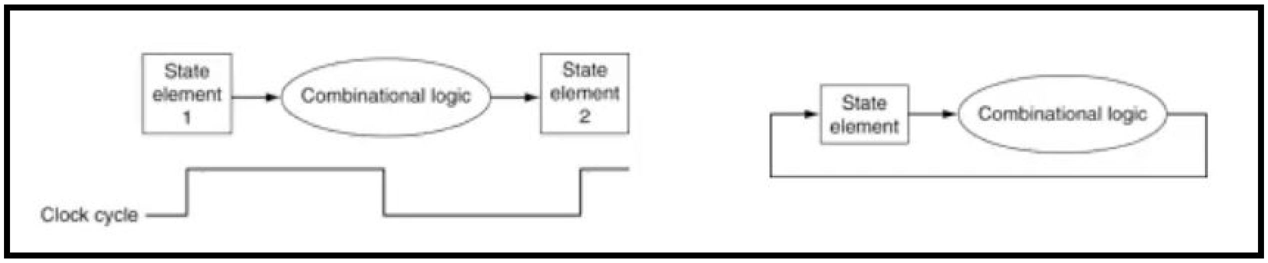
- Example of booths above

3_6 - Arithmetic for Computers

- For floating point values, they are represented with a sign bit, a fraction field, and an exponent field.
    - X = (-1)^S x (1 + Fraction) x 2^(Exponent - Bias)
- For most values, we want to normalize the fraction field so that it can fit more compactly.
- For the addition of 2 32 bit numbers, you have to take into account the formatting of these 3 fields.
- Floating point addition involves aligning decimals points, adding significands, normalizing the result, checking for underflow/overflow, and then rounding and renormalizing if necessary.
- Basically, there is a lot of additional hardware that is necessary to make floating point operations work, since you're dealing with a 32 bit format that contains different fields and those different fields have to be processed in different ways.
- Typically, floating point operations are multicycle, and can be pipelined.

4_1 - The Processor

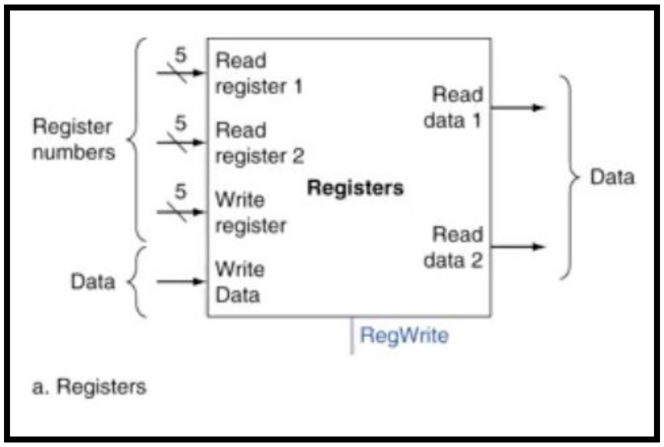
- There are two different MIPS implementation, single cycle vs pipelined.
    - In single cycle, each instruction takes a single cycle so CPI is 1, but the effect is that the cycle time will be determined by the longest latency instruction, so thus the clock rate will be pretty slow.
- Instruction execution involves having your instructions in memory, looking at where your program counter is pointing, and then fetching that instruction from memory. Then, depending on the instruction, we'll use the ALU to calculate something or access data for loads/stores or just set the PC to something different.
    - Remember that the PC is a 32 bit register that holds the address of the current instruction.
- On these diagrams, the black lines will show your data path and how inputs/instructions/data flow through the network, and the blue lines show the control structure which determines where that data should go and what operations should happen at particular ALUs, MUXs, etc.
- Voltage (High or Low) is encoded in binary. There is one wire per bit, and multi bit data is encoded on multi-wire buses.
- You have combinational elements that operate on data and the output is a function of the input. And you also have state elements which store information.
    - Combinational elements can be AND gates, MUXs, Adders, ALU
    - Sequential elements, on the other hand, need to store data in a circuit and thus they need to use a clock signal to determine when to update the stored value. You will basically update the state when Clk changes from 0 to 1. You can also have write control where you only update on clock edge when the write flag is 1.
- A common thing to do is to combinational logic transform your data during the clock cycles (so basically between clock edges).



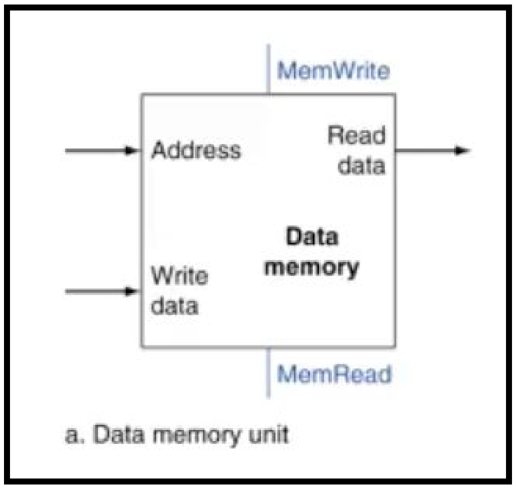
- Example above

4_2 - The Processor

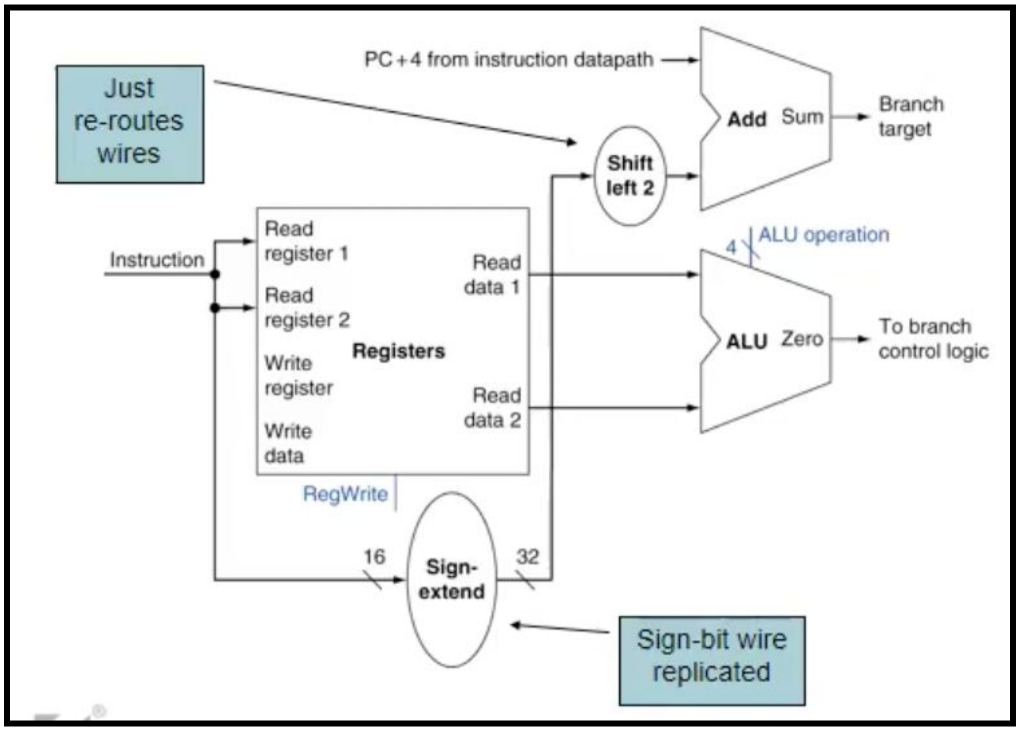
- The datapath is the set of elements that process data and addresses in the CPU. This involves registers, ALUs, MUXes, etc.
    - The control component is what determines how data flows through this and what operations get performed.
- In an instruction fetch, two things happen. We look at the address indicated by the PC and get that instruction, and we also use an adder to increment the PC by 4 for the next instruction.
- The R-type instructions basically read two register operands, perform some arithmetic, and write to an output register.



- RegWrite, btw, is a signal that tells us if we want to write or not at every cycle.

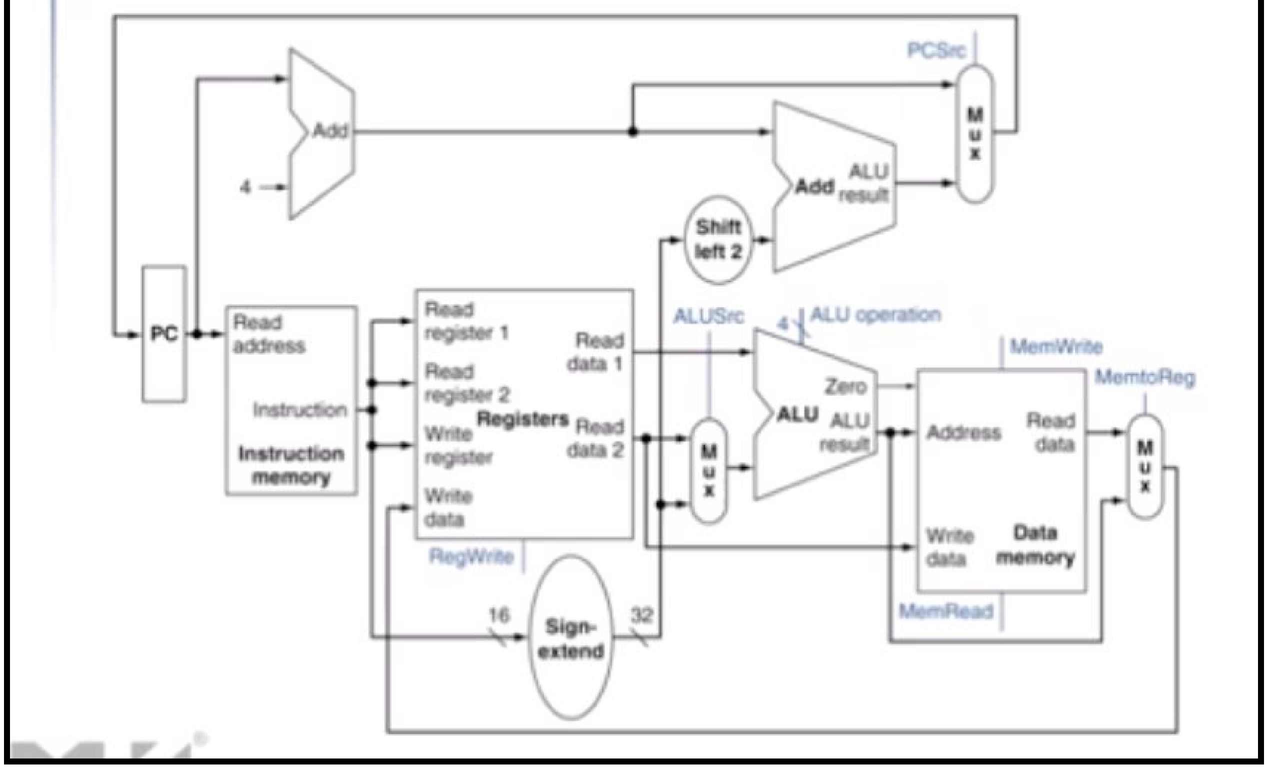


- With load/store instructions, we have an address that comes in (gets sign extended first) and depending on the operation, we'll either read memory and update a register or we'll write a register value to memory.

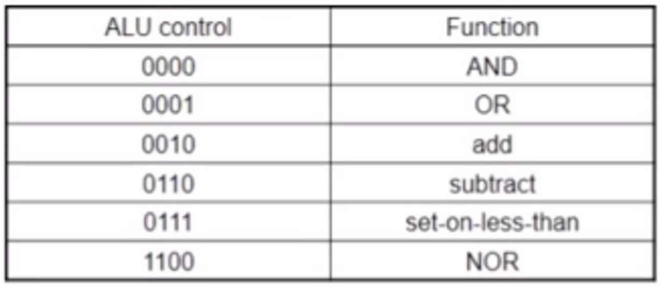


- With branch instructions, we'll need to read register operands and compare operands and calculate a target address by shifting 2 places, sign extending, etc.
- First cut data path does an instruction in one clock cycle and so each datapath element can only do one function at a time.
- In your datapath that will contain a combo of R-type/Load/Store, etc you'll have a lot of different components, but depending on your instruction, only some need to be used. Thus, that's where the control comes in and tells you which components you'll need to use.

Full Datapath



- The full datapath is shown above
- ALU can be used for multiple different instructions. It's used for loads and stores (F = add), as well as branches (F = subtract), and for R-type (function depends on the type of instruction we want to do - Could be AND, OR, SLT, etc).
    - There is also an ALU control which decides what the ALU will do at different times.
    - We do an add for loads/stores because we need to calculate a certain address.
    - Branch uses a subtraction because we need to do a comparison between two register inputs.

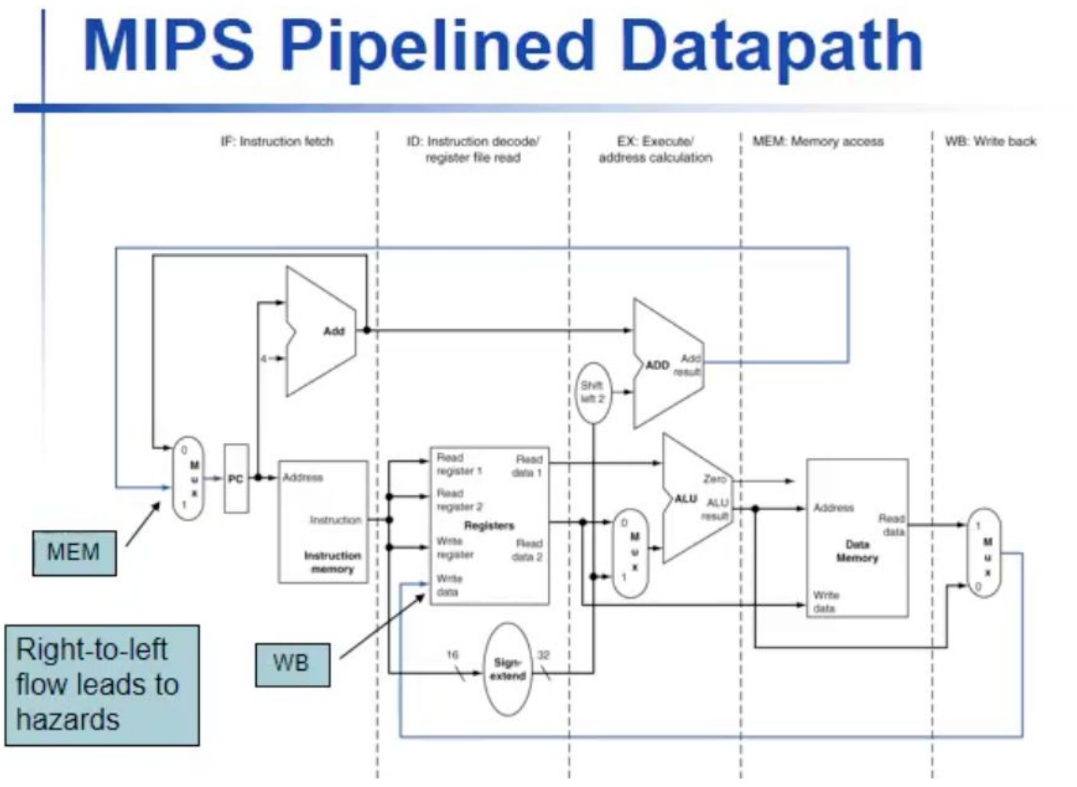


ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

- The ALU control will need to be able to take in some input bits, and then tell the ALU which operation to do.
- The ALU control will take in two inputs. One is ALUOp which is a 2 bit value (which is derived from the 6 bit opcode) that tells us if the instruction we have is lw, sw, beq, or R-type. Then we also have a function field, so that if we do have an R-type instruction, then we can specify further which ALU function we want.
- This single cycle implementation is not enough. Although it is relatively simple and the CPI is 1, the cycle time has to be extremely long (since every instruction has to complete in the time period).
    - And you may have different operations that has vastly different responsibilities, and thus can take different times to complete.

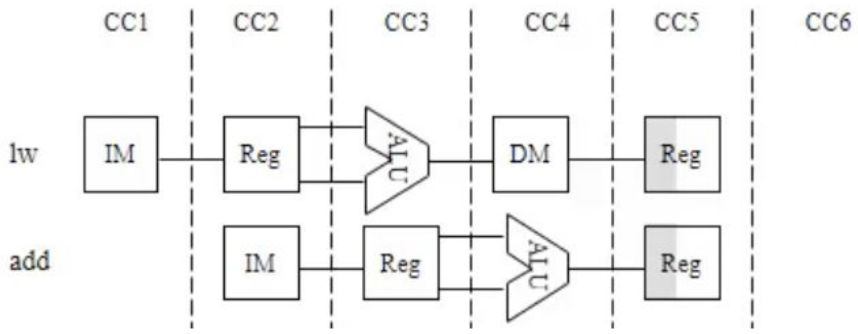
4_4 - The Processor

- Longest delay of any instruction in single cycle datapath will determine the clock period.
    - In our case we saw before, the critical path is the load instruction. And this might not be our most common instruction (might be R-type), so it's limiting us.
- Pipelining is all about overlapping execution. Basically, it's about finding the discrete steps that are independent, and thus can overlap.
- The MIPS pipeline has 5 stages: Instruction fetch (IF), instruction decode (ID), execute (EX), access memory (MEM), and write result to register (WB).
    - Difference is which ports and components are being used in the datapath.



4_5 - The Processor

- With pipelining the goal is to reduce the clock cycle time (which would be long af in the single cycle datapath)
- The maximum latency of any stage in a pipelined implementation will set the cycle time.
- If all stages are balanced(they all take the same time), then the time between instructions = time between instructions (for nonpipelined) / number of stages.
    - The speedup is due to increased throughput, not latency for each instruction.
- Different instructions, though, don't all need to go through the same stages in memory. A lw has to go through 5, while an add will have to go through 4.



- There could be a situation like above where we have two instructions at the same stage at once.
- MIPS is designed for pipelining since all instructions are 32 bits, few/regular instruction formats, load/store addressing, and alignment of memory operands.
    - RISC designs in general work great for pipelining.
- All instructions that share a pipeline must have the same stages in the same order.
    - An instruction can do nothing during a particular phase though.
- A main difference with the pipelined datapath is that we need to store information between stages and thus we have registers between stages to hold info from prev cycle
    - Data is stored at the rising edge of the clock.

4_6 - The Processor

- Any value you want to use at later stages has to be propagated along as well. You'd just have lines going from each register latch to each successive register latch.
- The value for write register, in particular has to go all the way to the Mem/Wb latch because it is only at that point where we actually have our write data, and it is only at that point where we can go back to the RF and perform the operation. That's why we have to propagate our write register value along.

4_7 - The Processor

- The control signals necessary for the different components in our datapath are derived from our instruction and then stored in our register latches and propagate as needed.

4_8 - The Processor

- Hazards are situations in the pipeline where the performance will drop and give you CPIs above 1.0.

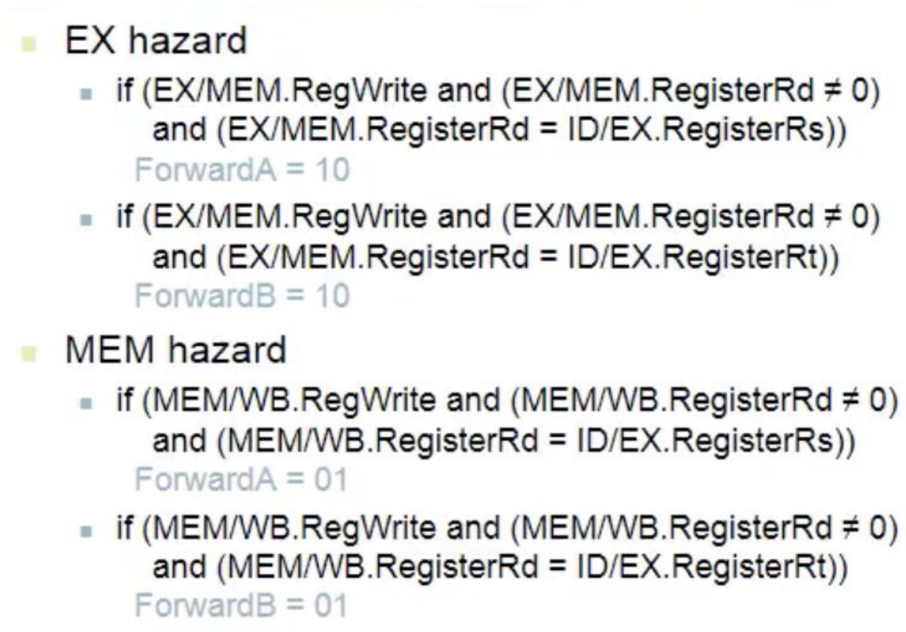
- Straight line dependence means that each subsequent instruction uses the output value from the previous instruction.
    - A data hazard will come up when we have this dependence. If we still try to do normal pipelining in these cases, then we will possibly be reading the not updated values for certain registers.
    - More specifically, the hazard is defined as one instruction that depends on another instruction is reading the incorrect value because the new value has not been written yet.
- Hazards are situations that prevent us from starting the next instruction in the next cycle.
    - Structure Hazard: A required resource is busy
        - 2 instructions can't use the same resource at once.
    - Data Hazard: Need to wait for previous instruction to complete its data read/write
    - Control Hazard: Deciding on control action depends on previous instruction.
        - Gotta wait on some sort of branch or conditional.
- Transparent latch is when you're able to write a value (this is normally dealing with the register file writeback stage) in the first half of the cycle and it would allow us to read that same register in the 2nd half of the cycle.

4_9 - The Processor

- To deal with data hazards, we insert independent operations (in software) and for dealing with them in hardware, we stall the pipeline by inserting bubbles and/or data forwarding.
- If the hardware just expects the software to put no-ops in between dependent instructions, that is a pretty big burden on the software side and this means that there is a lack of portability since you don't know what type of pipelining a machine architecture may implement and thus, you'd never know how many no-ops to put.
    - But on the other hand, no added complexity to the hardware
- Software can either insert no-ops or other instructions that don't use the register so that data hazards don't occur.
    - While CPI and CT will stay relatively the same, the IC will increase a lot.
- Another software option is to reorder the code to avoid use of load result in the next instruction, but this is hard af with loads/stores and branches.
    - An example is to put 2 load words in succession and then 2 add operations rather than one load word, stall, then add, and repeat again.
- Stalling the pipeline is basically the hardware equivalent of adding no-ops.
- If you stall one instruction, you have to also stall all the instructions behind it.
- To implement stalls, we have to first detect the hazard, and then stall the pipeline by preventing IF and ID stages from making progress since we don't want to lose information about the instruction.
    - We don't want to write to the PC and we don't want to write to the IF/ID registers. We also need to set all control signals propagating to Ex/Mem/Wb to 0.
- To see if a stall is going to occur, we need to check RegWrite and if RegDST matches one of the registers being used.

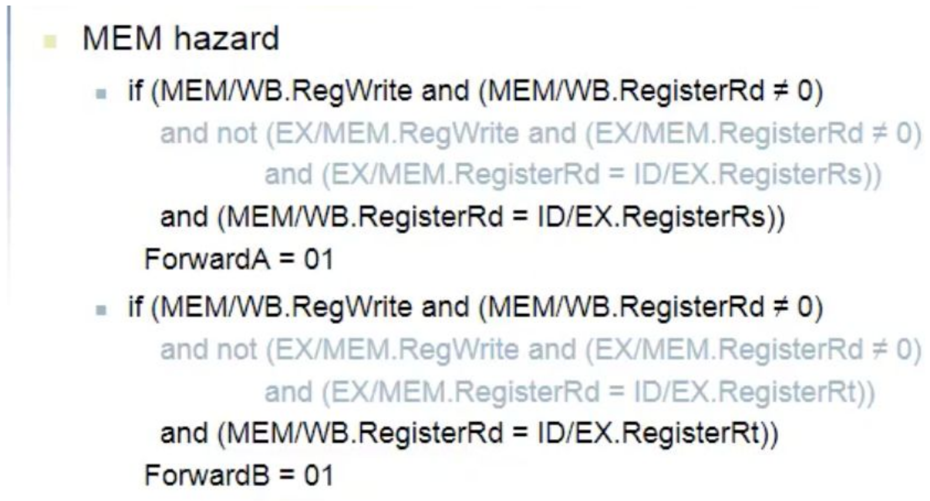
4_10 - The Processor

- Another way to handle data hazards is with forwarding or bypassing.
- The idea is that even though we haven't written the new value of something back to the register file, it might be available earlier than the writeback stage.
    - For example, it might be ready at the end of the execution stage.
- We are basically forwarding the value to the next instruction instead of going through the process of going through memory and the whole writeback stage.
- If a value is ready after it has gone through the ALU, then that means it will be available in the EX/MEM latch. The register value itself will not have gotten changed yet, but the new value is there in the latch.
    - Remember that in forwarding, the value is ready after the EX stage, and can be passed to the beginning of the EX of the next instruction. Therefore, if the instruction you're currently forwarding from is an R type, and you're forwarding to the next dependent instruction, you don't have to stall at all.
- Data hazards will be detected when
    - Ex/Mem Register Rd = Id/Ex Register Rs
    - Ex/Mem Register Rd = Id/Ex Register Rt
    - Mem/Wb Register Rd = Id/Ex Register Rs
    - Mem/Wb Register Rd = Id/Ex Register Rt
- Basically the above hazards check for whether the next instruction (which is in the ID/EX latch currently) has a register that will conflict with a register that is in either the Ex/Mem or Mem/Wb latches.
    - We also need to check if anything is even getting written. That gets determined by the Ex/Mem RegWrite as well as the Mem/Wb RegWrite.
    - Last criteria is making sure the Ex/Mem Register Rd and Mem/Wb Register Rd is not 0 either (because that is a no-op).



- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

- The forwarding unit is the piece of control that determines which values we want to proceed with.

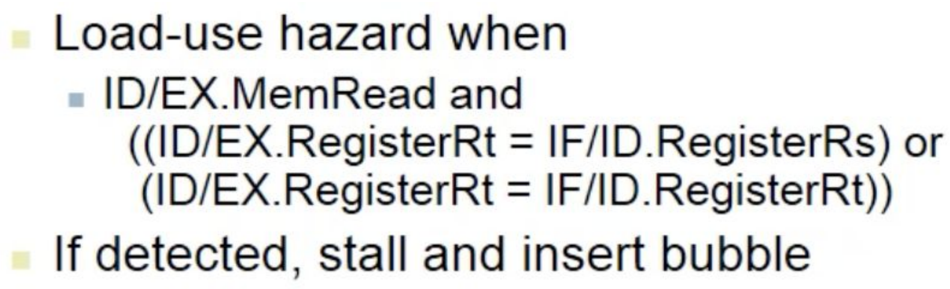


- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

- Double data hazard means that the EX hazard has precedence over the Mem hazard.
- In forwarding, you can forward from either the EX/MEM latch or the MEM/WB latch.

4_11 - The Processor

- Unlike R type instructions where the value is ready at the end of the execution stage, there are some instructions like load word, where the values isn't necessarily computed at that time. It will only be ready after the Mem stage.
  - Thus, there sometimes isn't an alternative to just stalling.



- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

- Thus, we want the waiting instruction to stall in the ID phase so that we can set all the control bits in the ID/EX register to 0 in order to have that stall.
  - We also want to prevent the update of PC and IF/ID register.

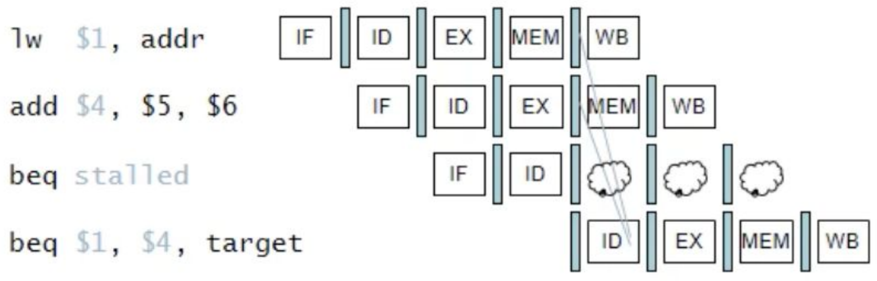
4_12 - The Processor

- Hardware stalls reduce performance but are required for accuracy. The compiler can try to arrange the code to avoid these hazards and stalls, but it needs knowledge of the pipeline structure, beyond just the ISA.
- The tradeoff between these two comes in static vs dynamic optimization.
- Control hazards arise because we have changes in control flow. A common case is with branches, where the next instruction depends on the branch outcome.
  - In data hazards, one instruction depends on another because of the values in the data.
  - In control hazards, one instruction depends on another because of PC.
- Basically, the problem is that if the branch instruction is still in the ID phase, what do we fetch for the next instruction to go into the IF phase. There are going to be two possibilities based on whether the control takes the branch or not.
- Possible hardware solutions include stalling until you know the direction that the branch will go. You can also guess the direction and start executing but then you need to be able to undo, and you can reduce the branch delay.
- Static branch prediction is a guess based on instruction type
  - If you have an if statement you have a forward branch, and then backward branch for for loops. We predict backward branches taken and forward branches not taken.
- Dynamic branch prediction is a guess based on execution history.
  - Maintain extra hardware structures that determine whether or not a branch is likely to be taken based on program behaviour. Table is updated as program runs.
- On important thing to note is that stalling has a fixed cost whereas prediction has a variable cost that depends on how often we are wrong and what the average cost is every time we are wrong.
- For stalling, we basically just wait until the branch outcome (during the MEM phase I believe) before we fetch the next instruction.

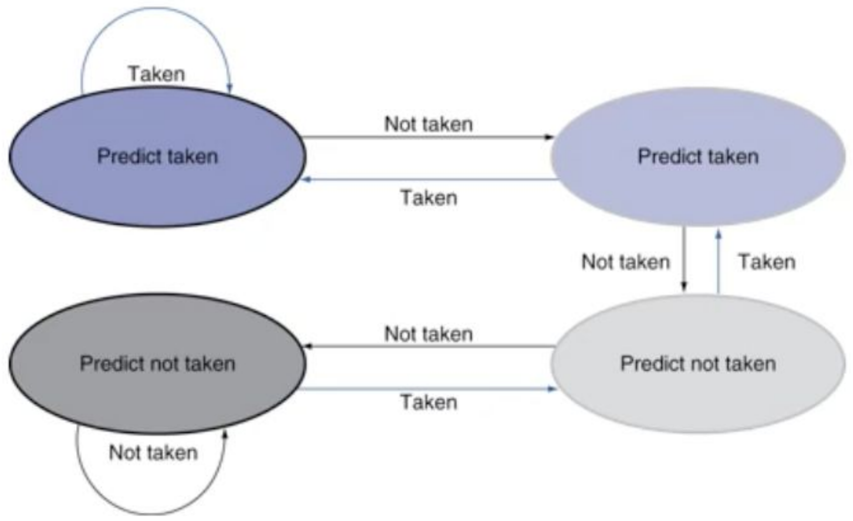
- With prediction, we only stall if our prediction is wrong. If it is wrong, then we have to flush by converting the instruction(s) into a bubble. We have to turn those instructions that we started to execute into no-ops (set control bits to 0).
    - Those instructions will not have done anything to data memory or to the register file so setting the control bits to 0 will be sufficient to make sure that no state was updated.
- Reducing branch delay involves moving hardware to determine outcome into the ID stage. We thus would need a target address adder and a register comparator.

4_13 - The Processor

- One of the issues with the solution to reducing branch delay is that the register values may not have been updated with the correct values depending on the previous instructions before the branch instruction.
    - If the beq is preceded by an add and the beq is dependent on the outcome of that add, then there has to be a one cycle stall before the updated register value in the Ex/Mem latch can get forwarded to the ID stage of the branch instruction.



    - On the same note, we would need a 2 cycle stall if we have a load word instruction that precedes the branch, since the load word needs to go into the memory stage and the value has to get forwarded from the Mem/Wb latch.
- We can also try to just improve branch prediction. We can do so by using a branch prediction buffer that is indexed by recent branch instruction addresses, and that stores outcome (basically a 0/1 prediction for taken/not taken). To execute a branch, we would want to check table, follow whatever prediction it says, and start fetching from fall-through or target, and then if we are wrong, then we flush the pipeline and flip the prediction.
    - To create the actual values for the table, we'd want to set a particular branch PC to 1 if it was taken in the past, and 0 if it was not taken in the past.
- Some of the problems with trying to have good predictors is that your predictions could get messed up with branches that change a lot (even/odd problem) or if you have nested loops.



- One of the solutions is to have a 2-Bit prediction where you have 2 bits to represent taken or not taken, and so we only change prediction on two successive mispredictions.
- Let's say that our predictor says that we should branch. In that case, we need to calculate the target address which means that we will have a one cycle delay. To help reduce this, we create a branch target buffer which is basically a cache of target addresses.

4_14 - The Processor

- Exceptions and interrupts are unexpected events that require a change in the flow of control. An exception arises within the CPU while an interrupt comes from an external I/O controller.
- In MIPS, exceptions get handled by a System Control Coprocessor which saves the PC of the offending instruction, saves the indication of the problem, and jumps to a particular handler address where the software will try to figure out what happened.
- Another approach is that you have a different address that you just to depending on what type of error you have. The instructions themselves will either deal with the interrupt or they will jump to the real handler.
- The handler needs to read what the cause was, determine if needs to restarts, terminate, or report the error using EPC.
- Exceptions can be thought of as another type of control hazard where we are having to jump to some other address.
- In cases where we have an exception, we want the previous instructions to have completed, we want to flush the subsequent instructions, set cause and EPC register values, and then jump to the handler address.
- When you have an exception that can be restarted from, the pipeline can flush the instruction, the handler executes, and then returns to the instruction. The PC is saved in the EPC register.
- This exception handling is very similar to what we do for branch, except the offending instruction is also flushed.

- When dealing with multiple exceptions, we want to just deal with the exception from the earliest instruction and then flush the subsequent instructions.

4_15 - The Processor

- To increase instruction level parallelism, we can use a deeper pipeline which means less work per stage and more stages and a shorter clock cycle and could result in more hazards though.
- Another approach is use multiple issue which means that we have more than one instruction in each stage. The idea is that we want to replicate some stages or expand the stages so that we get more instructions going at once.
    - In these cases, the CPI could be below 1.
- The downsides to multiple issue is that the hardware is hard to make because we need the IF stage to get more than one instruction, we more ports on the register file, etc.
- Static multiple issue is where compiler has to group instructions (into issue packets) and compiler has to detect and avoid hazards. Problem is that you're now restricted to a particular type of machine and you have less portability.
- Dynamic multiple issue is where CPU examines instruction stream and chooses instructions to issue at each cycle and we try to resolve hazards using runtime techniques.
- We can use similar approaches to branch prediction with something called speculation which is the same idea except you allow instructions to impact the state of the processor which doesn't really happen with branch prediction.
    - You need a mechanism to roll stuff back though.
- Compiler can also reorder instructions and they can also include fix up instructions to recover from incorrect guesses.
- Hardware can also look ahead for instructions to execute and buffers results until they are needed.
- When an exception happens, in
    - Static speculation: We can have ISA support for deferring the exceptions.
    - Dynamic speculation: Buffer the exceptions until instruction completion.
- In Static multiple issue, the compiler needs to know about what particular instructions the hardware can handle in a particular stage, and that's how it will know how to package the instructions together.
    - You can think of the bundle of instructions as just one really long instruction.
    - The hard part here is finding independent instructions that we can bundle together.
- In the above approach, most of the time we assume that we can execute an ALU/branch instruction as well as a data transfer instruction.

4_16 - The Processor

- Loop unrolling is a process that we use when we use static multiple issue. The purpose is to unroll the loop body in order to find more parallelism so that we can have less no ops.

    - One of the techniques is to use different registers per replication. This is also called register naming.
- Unrolling a loop 3 times means that we have 4 copies of the loop body and we end with the one bne/beq.
- The main way we can reduce the number of instructions is to look for addi's that we can combine. One thing to remember though, is to remember to fix the displacement offsets.
- Another big thing that we can do is use more registers so that we can for example do two load words at the same time. This does take more memory, but that's cool.

4_17 - The Processor

- With dynamic multiple issue, superscalar processors are where the CPU has to decide whether to issue 0, 1, or 2 instructions in each cycle and it has to know how to avoid hazards, but we avoid the need for the compiler to schedule instructions.
- Static multiple issue is more used for long based optimization.
- In dynamic, we can execute instructions out of order to avoid stalls, but we have to commit the results to the registers in order.
- We do instruction fetch in order, we do the execution out of order, and then we do the commits in order.
- A reservation station is where an instruction can wait before we have the necessary available hardware to execute it or when their register operands are ready.
- With speculation, we predict branch and continue issuing, but we don't commit until the branch outcome is known.
- The reason we don't let the compiler do all this stuff is because not all stalls are predictable, we cannot always schedule around branches, and different implementations of an ISA have different latencies and hazards.
    - For the last point, we would have to change our compiler every time we changed our machine organization.

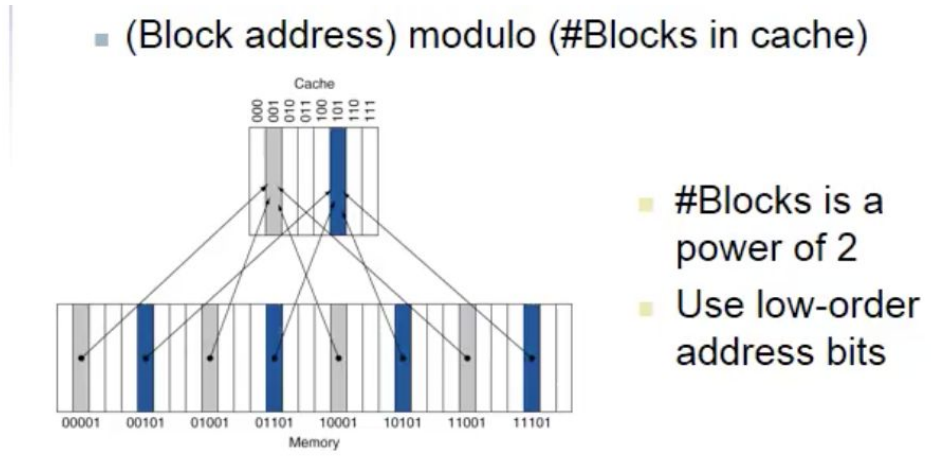
5_1 - Large and Fast: Exploiting Memory Hierarchy

- Different types of memory for computing.
    - Static RAM (used in caches): Fast but also really expensive in dollars
    - Dynamic RAM (used in main memory): Slower but less expensive.
    - Magnetic disk: Slowest, but least expensive.
- The ideal memory would have the access time of SRAM, with the capacity and cost/GB of disk.
- DRAM is implemented as a single capacitor per bit. It is also volatile so when you lose power you lose your data. It is in the middle in terms of latency and capacity.
- DRAM has scaled in capacity and cost has gone down but the latency hasn't improved as much, which called the memory wall problem.
    - Getting memory to the CPU faster is the bottleneck that we're dealing with.
- Flash storage is nonvolatile and faster to access than hard disk, but it's more expensive and a little smaller.

- Disk storage has the highest capacity, and the lowest cost, but you'll have really long latency.
- The principle of locality is that programs access a small proportion of their overall address space at any point in time.
    - Temporal locality: Items accessed recently are likely to be accessed again.
        - Examples are instructions in a loop.
    - Spatial locality: Items near those accessed recently are likely to be accessed soon.
        - Examples are sequential instruction access or array data.
- To take advantage of locality, we will copy recently accessed items from disk to smaller DRAM memory, and then copy more recently accessed items from DRAM to smaller SRAM memory.
- To access memory, you go through the different levels one by one and keep track of hits and misses on SRAM, DRAM, etc and then we make note of which entries we want to copy up.

5_2 - Large and Fast: Exploiting Memory Hierarchy

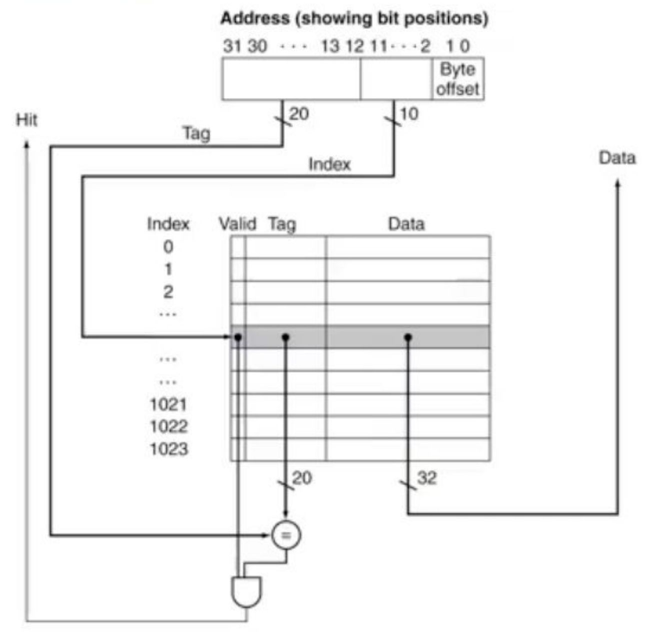
- Cache memory is implemented in SRAM.
- Cache memory is the level of memory hierarchy that is closest to the CPU, and we also have different levels of caches.
    - L1 would be the smallest size but the fastest



- (Block address) modulo (#Blocks in cache)
- #Blocks is a power of 2
- Use low-order address bits

- Direct mapped cache is the lowest overhead implementation of the cache. This is where the location is determined by address. If cache can hold 8 cache entries, that means we can hold 8 chunks of data.
- Cache holds a subset of memory and there has to be a way to figure out which of the 8 entries we want to look for, based on the address that we have.
- We also need tags and valid bits to know which block is stored in a cache location, and then bits will tell us whether there is any data present in a given location.
- Just like a hashtable, we're going to see problems whenever we deal with collisions. We may have to kick out values from the table even though we may have space in other locations.

Address Subdivision



- This shows how we do the addressing.

5_3 - Large and Fast: Exploiting Memory Hierarchy

- Larger blocks should reduce the miss rate, but larger blocks means you have fewer of them which could increase miss rate.
- On cache misses, we will have to stall the pipeline, fetch the next block from the memory hierarchy, restart instruction fetch or complete the data access.
- Write through says if I'm writing to my cache (through a store word instruction for example), then I write to main memory immediately to keep the value consistent everywhere.
- Write back says that if I write to the cache, I keep track of it (using a dirty bit) and then when the block is evicted from the cache, then we write its updated value to memory.
- We talked about read hits and misses. For write misses, we can either bring that block into the cache and then update it, or we don't fetch the block and we just write to the higher levels in the hierarchy.

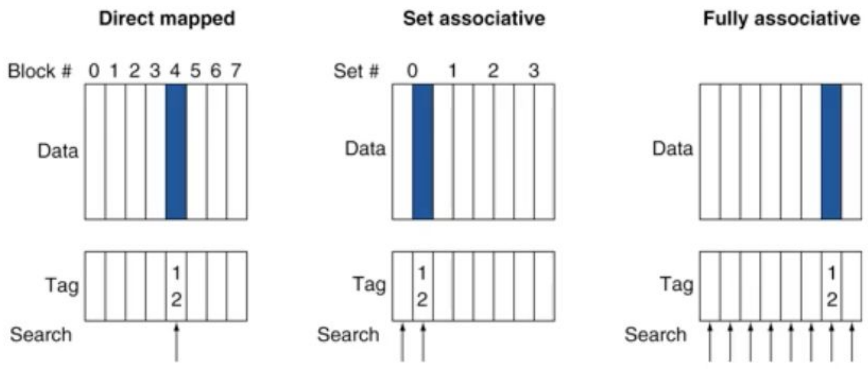
$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

- To measure cache performance, we have to think about how many misses we may have as well as what the penalty for each of those misses is.
- Average access time is the quantity that refers to the hit time.

    - AMAT = Hit time + Miss rate x Miss penalty

5_4 - Large and Fast: Exploiting Memory Hierarchy

- For direct mapped, there was only one particular location in the cache that we could have used to represent data memory. The other end of the spectrum is fully associative when you allow a given block to go in any cache entry and allow it to search all entries.
    - Downside is that we need a comparator per entry.



- Set associative is the middle group where the set has n entries and the block number will determine which set to go to, and then all the entries in the set will get searched.
- As n goes up, you have greater flexibility but more power and hardware needed.
- Increased associativity decreases miss rate, but with diminishing returns.
- When you're replacing values in the cache, you don't have a choice in direct mapped, but with set associative you can choose to prefer the non valid entries and then the least recently used or you can go random or FIFO.
- Smaller associative caches need to make good decisions on which values to kick out and which to keep.

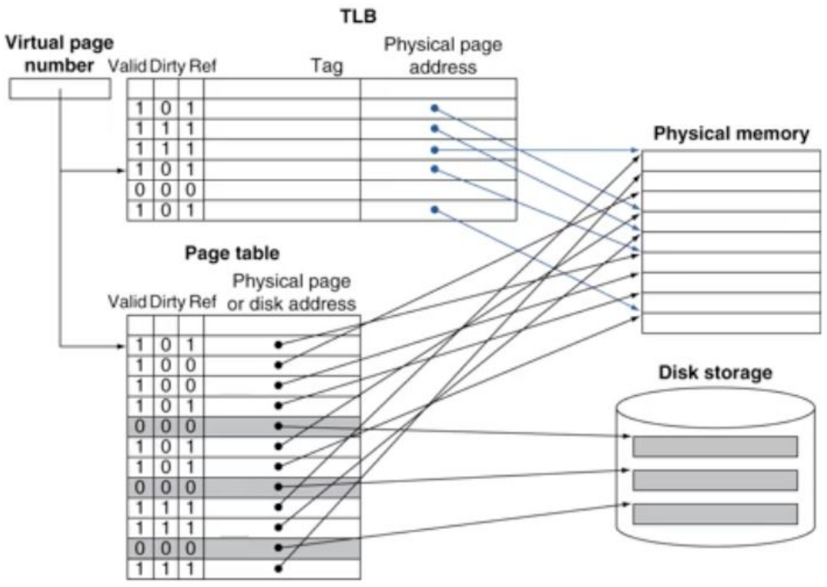
5_5 - Large and Fast: Exploiting Memory Hierarchy

- In multilevel caches, you have the primary cache attached to the CPU, and then the L2 cache handles misses from primary cache, and the main memory then handles L2 cache misses.
- Remember that if the problem gives you the penalty in terms of actual seconds instead of cycles, you have to convert it to cycles by dividing the penalty by the number of seconds per cycle which should be determined by the clock rate.
- In the primary cache, we want to focus on minimal hit time. For L2, we focus on low miss rate to avoid going to main memory. L1 will have a smaller block size than L2 block size.

5_6 - Large and Fast: Exploiting Memory Hierarchy

- Virtual memory is where we use main memory as a cache for secondary disk storage. It is managed by the CPU and the OS.
- Programs share main memory, and each gets a private virtual address space. The CPU and OS then handle translating those virtual addresses to physical addresses.

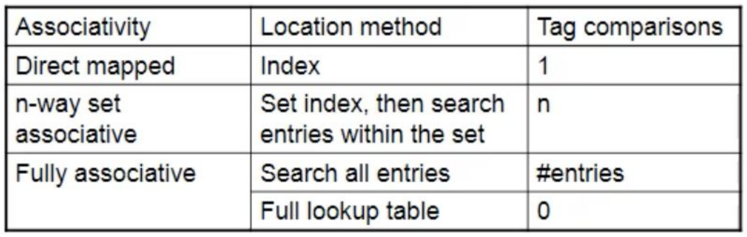
- Since each application thinks it has the whole address space, it doesn't have to care about which other applications are running at the same time.
- When you start to run a program, the linker will assign it a set of virtual addresses. During execution, we'll begin to map the addresses to physical addresses (and some will be in memory and some will be in disk).
    - Main memory acts like a cache.
- When talking about moving things in memory when it comes to address translation, we are using a page granularity.
- Each address (virtual or physical) will have an offset and a page number associated with it. Offset length is based on the size of the block.
    - Page offset will be kept the same when you translate from virtual to physical addresses. The upper bits could change though.
- When you're trying to access physical memory, and the value isn't there you go into a page fault, which means that the page wasn't in memory and therefore has to be grabbed from disk, which takes millions of clock cycles.
    - To minimize page fault rate, we use fully associative placement.
- The placement information of mappings is in a page table where the entries are indexed by virtual page numbers. The CPU will have a register that points to this table since this is a memory resident structure, not a structure that is in the hardware.
    - If page is present in memory, then PTE stores the physical page number
    - If page is not present, PTE can refer to location is swap space on disk.
- The valid bit associated with each entry in the page table refers to whether the entry is in physical memory (1) or if it is currently in disk (0).
- In these problems, we really really want to reduce the number of times we go to disk, and thus we will use write back when we want to modify something.
- To reduce page fault rate, prefer Least Recently Used replacement.
- The TLB is a cache of the page table. Basically allows you to exploit locality in the page table values. We use the TLB as a way of having to avoid going to the actual page table. It's just like any other cache.



- Basically, we first take the virtual page number and go to the TLB to check if it's there. If it is, then we can just look up the address in physical memory. If it is not, then we have to go to the actual page table, and we can also put it in the TLB for next time.

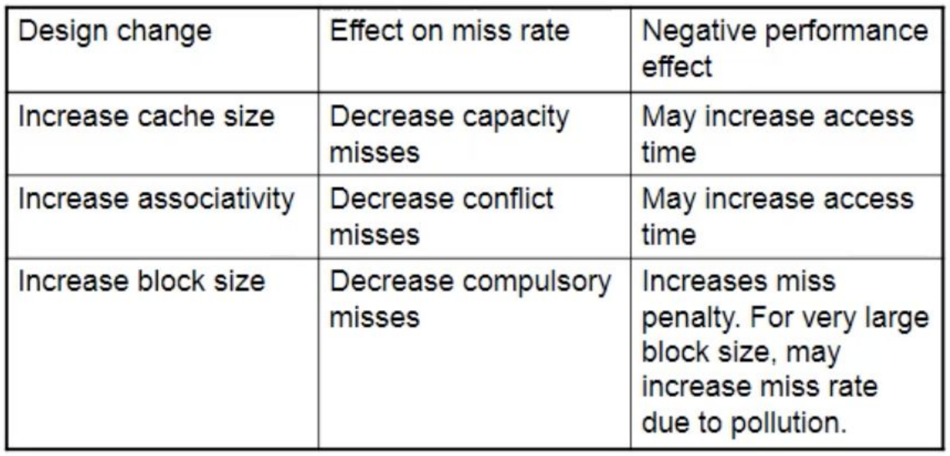
5_7 - Large and Fast: Exploiting Memory Hierarchy

- All of the memory hierarchy is based on the idea of exploiting locality using caching.
- At each level, there are different approaches to block placement, finding a block, replacement on a miss, and write policy.
    - Block placement is determined by associativity (DM, k-way SA, FA)
        - High associativity reduces miss rate, but increases the complexity, cost, and access time.
    - Finding a block is based on what associativity you use



Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

    - Replacement on a miss: LRU or random. In virtual memory, we can do LRU approximations
    - Write policy: Write through, write back, write allocate, write around
- 3 types of misses
    - Compulsory misses when first access to a block
    - Capacity misses due to finite cache size
    - Conflict misses where it wouldn't occur in a FA cache of the same size.

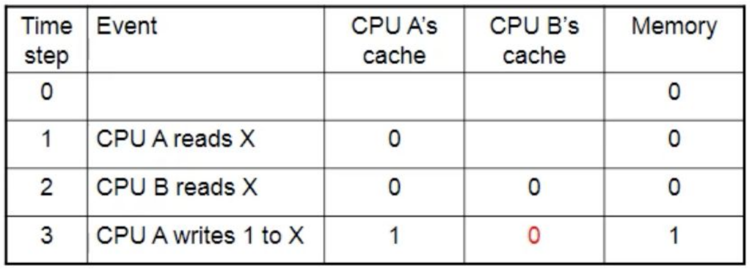


Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

- Cache design tradeoffs above
- Caching gives the illusion of us having a large and fast memory by trying to exploit locality.

5_coherence - Large and Fast: Exploiting Memory Hierarchy

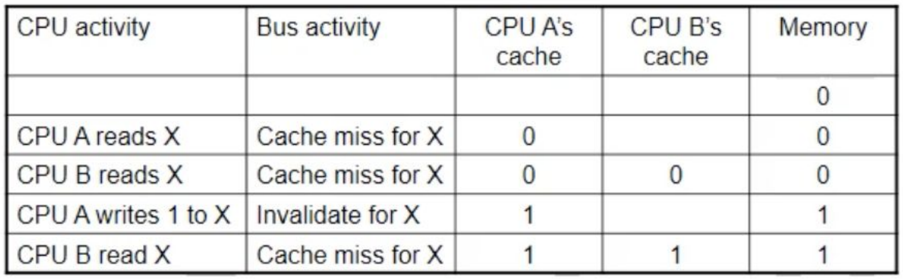
- Cache coherence problem deals with a multiple CPU case where a physical address space is shared, and each core/CPU has their own cache, but they have that shared memory. Thus, any change made to CPU A's cache needs to be seen by CPU B.
    - An example is when CPU A writes something, and the change is seen both in its cache as well as in the memory (since it is write through), but not seen in CPU B's cache.



Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

- Cache coherence means that the reads need to return the most recently written value.
- Synchronization from the software level doesn't solve this, you need architectural support as well.
- To ensure coherence, caches need to do migration of data to local caches and replication of read shared data.
- One of the cache coherence protocols is the snooping one where each cache needs to be on the lookout on the bus for reads/writes to the shared memory, and then update their cache accordingly.
    - The burden is on the caches themselves to maintain coherence
- Directory based protocols are where you have separate structures that record the sharing status of blocks in a directory. It tracks modifications and then forces the caches to update to the correct values.

    - Takes burden away from the individual caches.

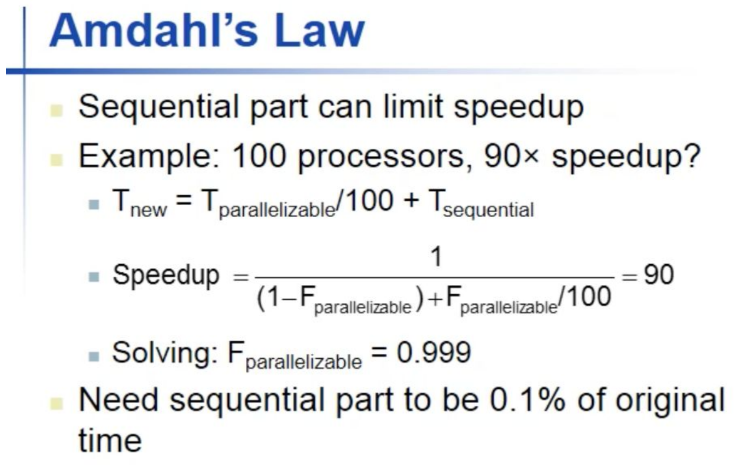


CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		1
CPU B read X	Cache miss for X	1	1	1

- Example of snooping protocol above. During the 3rd step, the protocol is basically telling every CPU besides A that your current value for X is not up to date and thus invalid and thus you should remove it from your cache. That explains why during the 4th step, we have a cache B miss for X, since it had removed the original value of 0 during the 3rd time step.

7_1 - Multicores, Multiprocessors, and Clusters

- Goal of multiprocessor is to connect multiple computers to get higher performance. We want job level parallelism (high throughput), parallel processing (single program on multiple processors), and multicore processors (chips with >1 processor)
- Solution is to add more workers instead of trying to make each individual worker hella efficient. There is also distributed reliability, so that even if one worker goes down, you still have the others to finish the job.
- Main challenges with parallel programming are partitioning (how to split work among threads), coordination (avoid race conditions between threads), and communication overhead between threads.



Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
    - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
    - Speedup $= \dfrac{1}{(1-F_{parallelizable})+F_{parallelizable}/100} = 90$
    - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

- Amdahl's Law when it comes to find if a particular speedup is possible with X processors
- Strong scaling refers to situations where the problem size is fixed. Weak scaling is where the problem size is proportional to the number of processors.
    - Useful for making sure we're comparing speedups in the right situations.

7_2 - Multicores, Multiprocessors, and Clusters

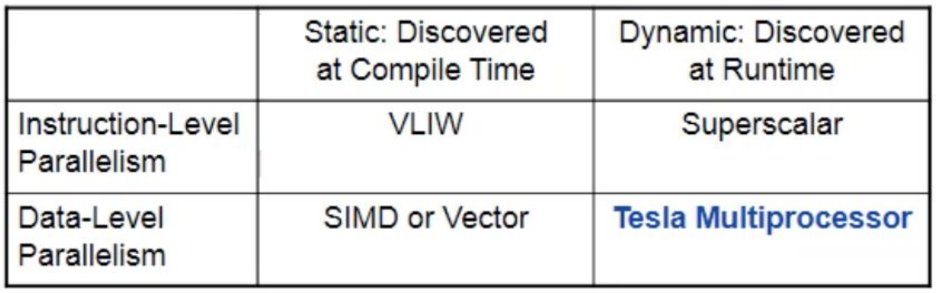
- For multiprocessor architecture, you could have a shared memory where there is a single address space for all processors, and we synchronize shared variables using locks. In terms of memory access time, we could see UMA (uniform) or NUMA (non uniform) between the times associated with each processor.
    - Con is that all the memory is in one place which could create bottlenecks.
- Synch() in the code is the part where the threads have to wait until all the threads have reached that point.
- Alternative to a shared memory approach is message passing where each processor has a private physical address space and the hardware sends messages between processors.
    - Con is the loss of shared variables and having to undergo some communication overhead.
- Loosely coupled clusters are an example of the message passing approach. Each computer has a private memory and OS, is connected to each other using I/O system (this provides means for communication), suitable for applications with independent tasks.
    - Con is that the interconnect bandwidth is relatively low. Any communication that occurs will have high overhead.
- Grid computing is like the above except on steroids. The separate computers are connected by long haul internet connections, and we can make use of idle time on PCs.
    - Works very well for tasks that require very little communication.

7_3 - Multicores, Multiprocessors, and Clusters

- C

7_4 - Multicores, Multiprocessors, and Clusters

- GPUs were created so that graphics computation could take place outside of the CPU.
- In GPU architectures, the processing is highly data parallel.
- GPUs are made up on SMs (streaming multiprocessor), and those are made up of SPs (streaming processor). Each SP is fine grained multithreaded.
- A warp is a group of 32 threads that are executed in parallel and in a SIMD style.
- GPUs aren't necessarily SIMD/MIMD because they have conditional executions in their threads.



	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**