CS M151B Notes

Yash Lala

Winter 2022 w/ Glenn Reinman

Supplementary Notes

- Questions for Reinman
- Lectures

Course Logistics

No electronic devices for tests except for calculators. Tests are open notes; print out these notes (+ Reinman's PDFs).

Chapter 1: Intro

Computers are everywhere, etc. You know this.

Current Trends in Architecture

Moore's law has ended – current trend towards specialized architectures (TPUs, etc). More diversity than we've seen in decades – *heterogeneity* is key.

PostPC era. Personal computers mainly dead. Have mobile devices that act as networked thin clients, and cloud devices.

Mobile devices now use one SoC (SoC := System on a Chip). Includes CPU, memory, modem, etc on one chip. No longer limited by a chip's connections to the board, but can have heat problems.

CPU consists of: - Datapath: Performs operations and transformations on data. Adds, loads, etc. - Control: Sequences the datapath, memory accesses, etc. - Caches

Abstract CPU through *ISA (Instruction Set Architecture)* (eg x86). ABI consists of ISA (user process runs on a chip) plus the system software's interface (syscall codes, etc).

Semiconductor Tech

Silicon ingot is sliced into giant blank *wafers*. They're patterned; each subsquare is tested for functionality. Saw the wafer into small *dies*, bond dies to a package, then send them to customers. Goal is to maximize *yield* (:= proportion of working dies per wafer). Every defect will cause the entire subsquare (becomes 1 die) to fail. So smaller dies $\implies$ defects propagate less $\implies$ higher yield.

$$\text{cost per die} = \frac{\text{cost per wafer}}{\text{dies per wafer} * \text{yield}} \text{yield} = \frac{1}{(1 + (\text{defects per area} * diearea/2))^2}$$

Performance

Performance is complicated, many metrics.

- *Response time*: How long it takes to do a task (to completion).

- *Throughput*: Total work done per unit time.
- *Performance*: 1/Execution Time.
- *Execution time can be decomposed*:
  - *Elapsed time*: Wall clock. OS time, I/O latency, process switching, all included.
  - *CPU time*: Time spent processing given job. We'll focus on this, neglect the outside system.

Usually, CPU has a clock. Asynchronous circuits use considerably less power, but are less fast. Check out the **Counterflow** paper for a cool design.

- CPU Time := CPU Clock Cycles * Cycle Time = CPU Clock Cycles / Clock Rate
- Clock Cycles = Instruction Count * Cycles Per Instruction (CPI)
- CPU Time = Instruction Count * CPI * Clock Cycle Time

Gotchas:

- Instruction count isn't just binary size. In a sense, if a loop is triggered 100 times, it counts 100x. Different senses of "instruction count", make sure your definition aligns with the speaker.
- Don't treat the # clock cycles as a given! Can improve it by improving the cycles per instruction. More performance doesn't need faster clock.

CPI is a weighted average. Weighted based on a workload – if 95% of the binary is an add instruction (10 cycles), then the processor CPI will lean hard towards 10 even if all other instructions take 1 cycle.

Big picture: Final Dimensional Analysis:

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Clock Cycles}}{\text{Instruction}} * \frac{\text{Seconds}}{\text{Clock Cycle}}$$

First is Instruction Count (IC). Second is CPI. Third is clock speed T_c.

Ahmdahl's Law: if you improve an *aspect* of a computer, the total performance won't increase proportionally. If we only spend 1 minute swapping, then speeding up swap can speed us up by 1 minute tops!

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

Corollary: Improve by making the common case fast.

Don't use MIPS (IC/Execution Time) as a metric. Instructions aren't all equal across architecture, instructions have different complexity.

Power

Now almost more important than performance. For CMOS ICs:

- Power = Capacitive Load * Voltage^2 * Frequency.

Problem with current generation: voltage doesn't scale down. Leakage problems. This is called the *Power Wall*. We can't reduce voltage more, can't remove heat more effectively, so need other tricks. Dynamic CPU voltage and clock scaling, etc.

SPEC CPU Benchmark Set: an industry standard spec used to measure performance. Run set of programs, take geometric means of speedup ratio (as compared to old processor).

Chapter 2: The ISA

Part 1: RISC Instructions

Instruction set: repertoire of instructions accepted by a processor.

Early computers were CISC (complex instructions). Memory was expensive, so more compact instructions were very useful. Gradually, RISC started to take off because it simplified processor design (power draw down), and

because finer grained instructions allowed for more reordering and parallelization. Now, CISC processors break instructions into RISC-like "micro-operations" internally.

We'll go over MIPS, a RISC architecture.

Most arithmetic operations look like `add a, b, c` (`a = b+c`); they take 3 operands, even those that might only need 2 (will ignore 1 operand). This simplifies the ISA implementation, all instructions look the same.

Where do we get the data for the operands (`b`, `c`) from?

Firstly, we could read from registers. MIPS has 32 32-bit registers. Stored in register "file"; not actually in memory, just used synonymously with "data store". 32 registers implies that every instruction needs 5 bits per register to specify which one it's operating on. Adds up quick.

Secondly, we could use memory. Can transfer data from memory to registers, then read from the registers. CISC does this in every instruction (remember the `leaq` style parentheses addressing), but RISC usually has explicit `load` and `store` instructions to get data to/from memory. Some complications when using memory.

- Memory is addressed by the byte, but we can only fetch processor *words* (instruction size, int size, etc; 32 or 64 bits) from memory. Every memory read has to be word-aligned (can't read from address 0x2, only 0x0 (which will fetch 0x0, 0x1, 0x2, 0x3 as a single word)).
- MIPS is big-endian. MSB in the least address of a word.
- Don't confuse the instructions, don't flip the operands. `lw` loads *from* memory, `sw` stores *to* memory. Always relative to memory. `lw a, b` loads from address in register `b` into register `a`. Right to left. `sw a, b` copies the value of `a` into the memory address indicated by `b`. Left to right.

Lastly, we can store the data in the instruction itself. Suppose we want to add 3 to a number. The number "3" might be encoded in the instruction itself, which is usually called an *immediate* value (`addi $s3, $s4, 3`, etc). Saves space, but mind the range constraints.

Immediates only have a certain number of bits available, usually much smaller than word size (have to fit in instruction, eg `addi` gives us 16 bits). TLDR: Data can be encoded in instructions, too. Makes the common case fast, save a load.

How do we set a register to a constant? Most constants are small, can store then in 16 bits. If we need to set a register to a 32 bit value, then we use the `lui rt, constant` (load upper immediate) instruction, which sets the upper 16 bits of register `rt` to `constant` and clears the lowest 16 bits. Can then use the `ori` (`or` with a 16 bit immediate) to set the lower 16 bits. Terminology bad – `lui` *doesn't* load from memory.

MIPS also has a zero register, which is hardwired to `0` (assigning to it won't do anything). Why? We sacrifice some register space, but in return we don't need to implement a `mv` (register to register) instruction. Can just do `add $t1, $t2, zero`. Simplifies instruction decoding (all instructions are 3-operand), making processor more power efficient.

Part 2: Representing Integers

Mostly review.

- n bit unsigned integers can go from $2^n - 1$
- n bit signed integers (2's complement) can go from -2^{n-1} to $-2^{n-1} - 1$
- 2's complement: bit 31 is effectively the "sign" bit. Flipped sign as compared to unsigned.
- $-(2^{n-1})$ can't be expreseed in 2's complement. It'll just invert to itself.
- To negate a 2's complement integer, invert it (one's complement) and add 1.
- Sign extension: represent a signed number using more bits. Take the sign bit (msb), and replicate it to the left. Eg: `1010` goes to `11111010`. MIPS hardware does this automatically for some instructions; `addi`, `lb` and `lh` (load byte, load halfword) `beq` and `bne`, etc.

Part 3: Representing Instructions + Addressing Modes

MIPS instructions come in the form of single words. CISC architectures often have variable length instructions for compaction reasons, but it's not worth the decoding hassle for RISC.

We're using combinatorial circuit logic to work on an instruction. It makes sense to reuse these circuit elements as much as we can, so instructions tend to classify themselves into "formats". Very small number of formats $\implies$ easy to decode. MIPS has 3 main ones:

1. *R format* (R type): Deals with registers. The regular 3-operand instructions we've seen before, look like this:
   - `op`, the opcode. 6 bits. Not the instruction itself; just used to figure out the instruction's format. `000000` for R format instructions. Think of it as "formatting header".
   - `rs`, 5 bits. First register source.
   - `rt`, 5 bits. Second register source. (abcd. . . st).
   - `rd`, 5 bits. Destination register.
   - `shamt`, the shift amount. 5 bits. Not used for many instructions.
   - `funct`, the function code. 6 bits. Extends opcode; disambiguates all of Used for adds, subtracts, loads, stores, etc. Looks like `funct` has bits 0-5; so make sure to count in reverse.
2. *I format* (I type): Deals with immediate values. Alternatives to R format instructions, like `add` vs `addi`. Looks like:
   - `op`, the opcode. 6 bits. Different for every I format instruction; only rformats share the same opcode (because iformats generally very different and weird).
   - `rs`, 5 bits. First register source.
   - `rt`, 5 bits. Another register. Sometimes source, sometimes destination. . . iformats are weird instructions.
   - `constant` / `address`. 16 bits. A literal immediate, gives the iformat its name.
3. *J format* (J type): Used for long jumps (`j`, `jal`, etc). Encodes the full target to jump to as an immediate. Looks like:
   - `op`, the opcode. 6 bits.
   - `address`, the memory address to jump to. 26 bits. We can only jump to word-aligned addresses, so the `address` is expressed in terms of words (divide address by 4). Effectively can address 28 bits. But memory can be 32 bits. . . so just copy the upper 4 bits from the existing PC value, and set the lower 28 from the `address` field. Sometimes called "Pseudo Direct Jump Addressing". TLDR: jumps to $\mathrm{PC}_{31...28}..(4 * \mathrm{address})$ (where .. is concatenation).

These are specified in ISA, so all MIPS processors are compatible with these binaries.

Let's look at some MIPS bitwise operations:

- `sll` and `srl`: shift left and right (logical, ie pad with zeroes). This is an R format instruction that uses the `shamt` bits.
- `and` and `or`: also R format instructions. Ignore `shamt`.
- `nor`: `a nor b = not (a or b)`. We don't have a not instruction, so use `nor b, a, $zero` for `b = ~a`. Economy of instructions!

And some jumpy operations:

- `beq rs, rt, L1`: branch if equal. If `rs = rt`, then jump to `L1`, an address in the code itself. `bne` is the opposite; branch if not equal. How is a label encoded? We'll get to it later. Branch instructions are I type instructions. They jump to $(\mathrm{PC} + 4) + 4\mathrm{offset}$. Really, the $PC + 4$ part is implicit; all relative addressing usually assumes that `PC` has already been incremented. Why the $4 * \mathrm{offset}$? See below.
- `j L1`: Jump unconditionally to the label.

We use these to write if-statements, for loops, etc. We know the gist – all basic CS 132 stuff. Be aware that these operations receive their labels as *word* addresses, not byte addresses (because instructions must be word-aligned).

Basic Block: A chunk of instructions with no branch instructions (except at end) and no labels (except at beginning). Once you start, you'll always get to the end. Compiler optimizes in terms of these basic blocks. Advanced processors can parallelize executions within basic blocks (even between them in "hyperblocks").

More operations:

- `slt rd, rs, rt`: set less than. If `rs < rt`, then sets `rd` to 1. Otherwise, sets `rd` to zero. MIPS doesn't have flags like in x86, so this is the only way to do comparison (no automatic subtraction). `slti rd, rs, imm` is variant with immediate constant.

Why no `blt`, `ble`? The `<` and $\leq$? are very expensive. Would have to slow clock way down if we wanted it to be in the same instruction, so split it off into a separate instruction.

MIPS also doesn't have "greater than" style instructions. There are pseudo-instructions you can use in the assembly, but they're converted to equivalent `lt` style instructions under the hood anyways (either in assembler or on chip).

How do we call procedures? First, store parameters in registers. Save the caller saved registers, then jump. We've seen all of this in CS 132, don't rewrite it here.

MIPS has some dedicated registers and procedural instructions:

- `ra`: return address register
- `jal label`: jump and link. Jumps to label, and stores the "return address" (`jal` instruction's PC + 4 bytes) in the `ra` register.
- `jr ra`: jump register instruction. Used like a return statement, usually. Jumps to the register specified; here, copies `ra` to the PC.

These are all J format instructions. They can jump much farther than the `bne` instructions, because they have more bits for the address.

Broadly, when compiling functions, we can call them "leaf" procedures (don't call any other procedures) or "non-leaf" procedures (they call other functions). A non-leaf procedure has to save its return address on the stack, because when it calls the subfunction it doesn't want `ra` to be overwritten and forgotten. It also has to save callee-saved registers.

Summary of Addressing Modes:

1. Immediate addressing: Values are stored in the instruction.
2. Register addressing: Values are stored in register. Instruction specifies register.
3. Base addressing: `4(ra)`. Values are obtained by adding register value and an offset specified in the instruction.
4. PC-relative addressing: Values are obtained by adding PC to a (compacted) offset specified in the instruction.
5. Pseudodirect addressing: Long jumps. Values are obtained by a long compacted offset specified in the instruction, concatenated with a few bits from the PC.

Memory Layout

- Reserved segment lowest in memory.
- Above that is the Text segment, stores functions, etc.
- Above that is the Static Data segment, stores variables of global scope. Can access this with `$gp` register (global pointer).
- Above that is the Dynamic Data (heap) segment, evolves over scope of a program. Grows up.
- At the top is the Stack. Grows downward, into the void between the stack and the heap.

Chapter 3: Arithmetic for Computers

All instructions look like:

1. Fetch Instruction
2. Decode Instruction
3. Fetch Operand
4. Execute
5. Store Result
6. Next Instruction

We'll be focusing on step 4 this week. In particular, will discuss arithmetic operations as example of instruction execution hardware. Usually segregate arithmetic into integral and floating point components. Integral addition is handled by *Arithmetic Logic Unit (ALU)* chip component.

Some languages (eg C) ignore overflow, others require explicit exceptions. Luckily, addition algorithm is the same for signed and unsigned vals. So we have 2 versions of instructions; `addu` (unsigned), which ignores overflow, and `add`, which throws an exception on overflow.

What does a 32-bit ALU look like?

Inputs: - 2 32-bit inputs - 2 Opcode-selecting bits - 1 A-invert bit - 1 B-negate bit - The last 4 are called the control wires.

Outputs: - Zero Line: high if the overall result was 0. - Result: 32-bit result of calculation. - Overflow: a single bit to detect if an overflow has occured. Can go through some interrupt generating mechanism if needed. - Carry Out: The carry out bit from the last 1-bit ALU (MSB). We don't use it much (it's used to implement SLT, but that's it).

Carry out is *not* the same as Overflow. Overflow = Carryout in the case of addition, but subtraction overflow detecting is more complicated.

32-bit ALU is composed of smaller 1-bit ALUs. They take 2 bits as input, use basic gates (`xor`, etc) to calculate the addition, `or`, `and`, etc of the inputs (all in parallel). One "operation" input is used to multiplex between these output bits. There's also a carry input and output.

Can chain 1-bit ALUs in several ways:

Simplest is *Ripple Carry* Adder. Chain 32 1-bit ALUs, feeding the carry output of the LSB into the next bit, and so on. Very slow, carry has to propagate through all the adders. N bit ripple carry adder has $2N$ gate delays (every 1-bit ALU has 2 gate delays).

To reduce propagation delay, use *Carry-Lookahead* Adders – trade more logic (power++) for more speed. At the beginning of the computation, we know all of the input bits. So in parallel, we can compute whether each bit pair will *generate* a carry (`and(ai, bi,)`), or *propagate* a carry (`xor(ai, bi)`, ie a carry coming in will come out). This info depends only on the given bits, and can occur entirely in parallel. We can use this info to figure out the carry bits without waiting for carry bits to ripple through the entire adder. Say C_1 is the carry bit coming into ALU 2 ("out of" bit 1).

$$C_1 = G_0 \vee (C_0 \wedge P_0) C_2 = G_1 \vee (G_0 \wedge P_1) \vee (C_0 \wedge P_0 \wedge P_1) \dots$$

This looks stupid; because we're "brute forcing" over a large number of bits. But because all of the G_i and P_i are available after only 1 gate delay, we can compute the overall expressions above in parallel (there are no dependencies). Consider C_2; assuming a linear fan-in delay (3-bit `and` takes 3 seconds, 4 bit takes 4 seconds), then the delay is:

$$jC_2 = 2 \vee (2 \wedge 2) \vee (0 \wedge 2 \wedge 2) 2 \vee (4) \vee (5) 5 + 3 = 8$$

TIP: Remember that everything is in parallel when calculating delays! Choose only the longest dependency!

The overall delay can be shorter than a ripple carry adder. You can even have hybrid approaches, constructing a 32 bit adder out of 4 8-bit carry-lookahead adders arranged ripple-carry style. Be wary when assembling adders like this! Remember – the generate and propagate inputs depend only on their respective bits. They specify whether those bits generate or *could* propagate, not whether they actually do. Minimize data dependencies on every step. That's the insight of carry lookahead.

Now that we have a 32-bit ALU, how should we handle more complex interactions?

How do we handle subtraction? $7 - 3 = 7 + (-3)$. -3 is `~0b11 + 1`, so add a `binvert` (binary invert) input to the addition ALUs that inverts one of the input bits (`~`). Simulate the `+1` by triggering the carry input of the least significant 1-bit ALU.

How do we handle the `SLT(rs,rt)` instruction (`1` if `rs < rt`, else `0`)? Set all but the LSB of the output to 0. Now, subtract the two numbers and use the carry flag of the output as LSB of the output.

TODO: Talk about carry select

Now we want to add multiplication capabilities to the ALU.

Naive implementation is straightforward; multiplication is performed via the same algorithm we'd use in middle school.

```
     1011
*    0101
---------
     1011
    0000
   1011
+ 0000
---------
result
```

The top number is the multiplicand, the bottom is the multiplier. Store them in registers, then move as follows:

1. Right-shift the multiplier.
2. If the lowest digit of multiplier is 1, then add the multiplicand to the result register (64 bit).
3. Left-shift the multiplicand.
4. GOTO 1 (32 times)

We can make this more efficient by swapping the 64 adder for a 32 adder. Always add to the upper 32 bits of the result register, and then shift the result right (rather than shifting multiplicand left, just shift the result right).

Booth's Algorithm

Can do signed multiplication efficiently with Booth's Algorithm. Analogous to:

$$\sum_{i=1}^{n} 2^n = 2^{n+1} - 1$$

Look for a range of consecutive `1`s in the multiplier. Ordinarily, we'd have to add the multiplicand those many times, right shifting every time (aka multiplying by 2! Exponents!). Rather than perform all of those adds, we record the starting power (minus 1), shift to the most significant one, shift even further beyond, add the multiplicand, and subtract the multiplicand at the starting shift.

TLDR: Keep shifting right, and remember the most recently deleted bit (b)

- 0 (last was 0): no-op
- 1 (last was 1): no-op
- 1 (last was 0): subtract
- 0 (last was 1): add

Floating Point

FPUs require significant extra hardware, and take many cycles (because IEEE format is packed). High level algorithm involves unpacking it, aligning the mantissas based off the decimal point difference, using an enormous ALU to deal with the mantissas, renormalizing, and reencoding.

Chapter 4: The Processor

This is Chapter 4; the processor.

We'll look at 2 MIPS designs: one that runs 1 instruction per cycle always, and another that's pipelined. Only talk about a subset of instructions. Midterm questions will usually involve adding another instruction to this subset, and seeing how it affects the other instructions.

Then, we'll talk about problems with pipelining; namely data and control hazards. We'll finish with a discussion on how to make pipelined processors superscalar (>1 CPI).

MIPS Design 1: Single-Cycle Processor

CPU Overview

Look through the CPU Overview diagram in the slides. Great stuff.

Basics of Logic Design

How do we regulate data with the clock? Need state elements:

- Register: Has a data and clock input. On a rising edge, sets output to input. Sustains that output for the duration of the clock cycle, regardless of what happens to input.
- Write Controlled Register: Has an additional "control" input. On a given clock rising edge, only updates the output if the "control" input is high.

Use these to make synchronous processor. Store data in state elements, pass through combinatorial logic, and output to next state element.

Building a Datapath

Datapath := elements that process data and addresses in the CPU (eg: how data moves from registers, into ALU, etc). We'll cover all components here. Look at the slides, they're much better.

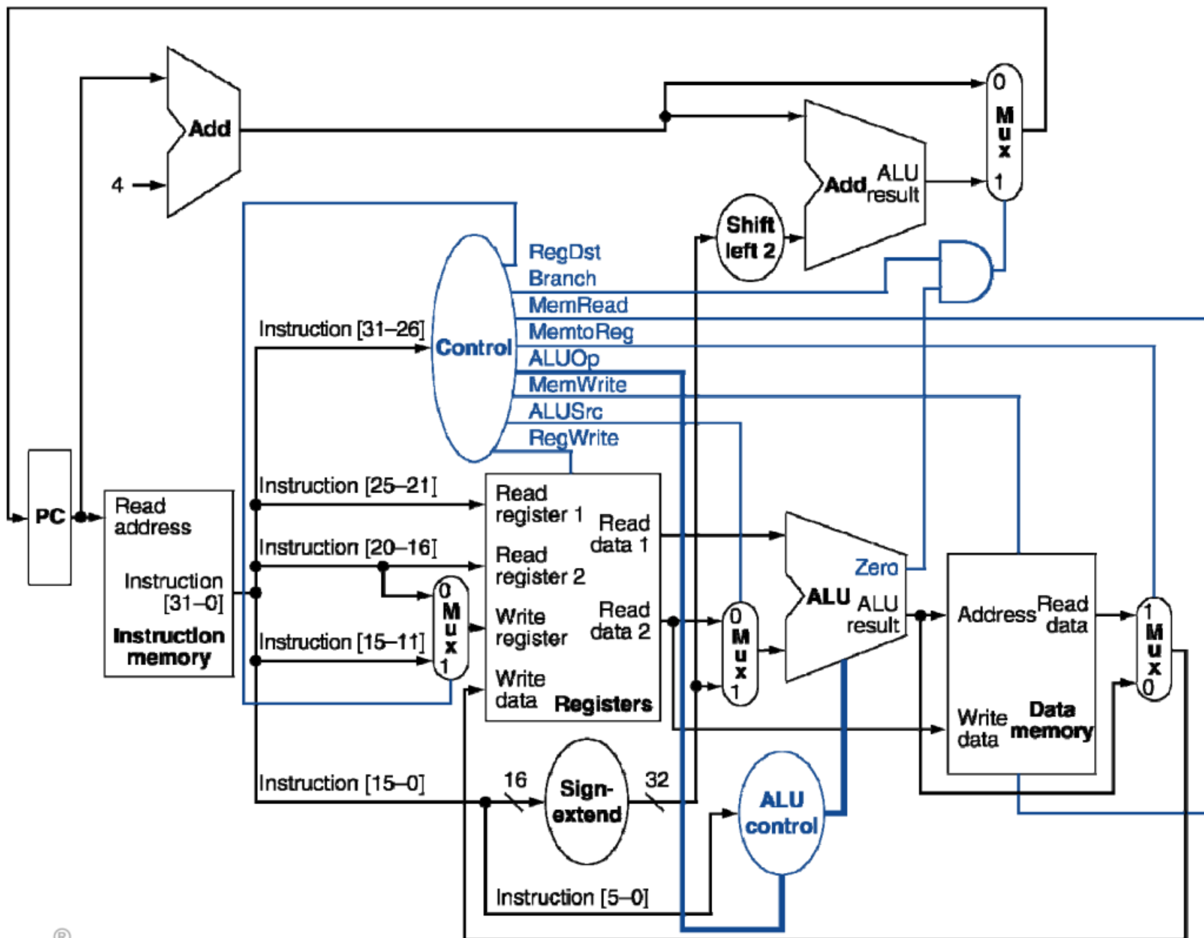

Figure 1: Single Cycle Datapath with Control

Some points:

- Instruction and Data memory is kept separate if we want to do everything in a single clock cycle (because we can't have a single memory component output 2 things at the same time).
- Instruction and data memory is *not* a clocked component, treat it as combinatorial. The moment an input goes into instruction memory, the value is produced as an output.
- We have 2 ALUs for things like `beq`. The main one (which we use for `add`, etc) is used to subtract. Its zero flag is used to figure out whether the condition was successful. Another ALU is used to add an immediate to the `PC+4` value, which allows us to execute the jump.

He then shows us the final datapath (we only covered how to do R-type instructions, but the diagram is full).

Building the Control Path

There's no good way to write this. Look at the slides, or the diagram embedded above. Some reminders below:

The data path (eg: ALU) requires several complicated control flags for a given opcode. We encode this compactly in the opcode; `0001` maps to add (set ALU add code), `0010` sets subtract (set ALU add code, set ALU B-negate, etc).

Looks like `lw` actually adds to an offset specified in a register. It's an I type, although this isn't specified in the assembler. I assume you usually use the stack pointer as the base.

- ALUSrc: If 1, use immediate for `y` of ALU. Otherwise use register file. Does instruction *have* a data source for ALU?
- MemToReg: 1 means read from memory, 0 means read from ALU. Writes the result to register file.
- RegDST: 0 means use `rt` as the write address for the register file (for I type). 1 means use `rd` instead, because R type has dedicated `rd`. 1 is there's an explicit destination register.

The opcode has a 2-bit ALU Operation Code section. It's used because several I-type instructions require use of the ALU (`beq` needs subtraction, etc). This ALU Op looks like:

- `00`: ALU should always add. Used for `lw`, `sw` (which add offset to the stack pointer).
- `01`: ALU should always subtract. Used for `beq`, which needs to compare two register inputs.
- `10`: ALU should defer to the `function` field. Only set for R-type instructions, because field only exists then. Quick way to switch "what field to look at".

MIPS Design 2: Pipelined Processor

Split the instruction execution into 5 stages.

1. IF: Instruction Fetch. Reads instructions from memory.
2. ID: Instruction Decode. Decodes instruction, reads from the register file.
3. EX: Execute/Address Calculation. Takes care of ALU, `PC+4`.
4. MEM: Memory Access.
5. WB: Write Back. Writes to register file.

Pipelining will have instructions in each stage at a given point in time. General Thoughts: - Pipelining doesn't change CPI if pipeline is full. Throughput increases, but not latency. Throughput increases because better resource utilization. - Allows us to lower our clock speed to match the slowest *component*, not the slowest instruction. - RISC design is great for pipelining.

Pipeline Principles: - All instructions sharing a pipeline should have the same stages in the same order. - All intermediate values are latched every cycle. Registers between each stage (eg: `IF/ID` is between instruction fetch and instruction decode). - Pipelining restricts functional component reuse (might be in use by earlier stage), so you need more ALUs, etc.

Pipelining TLDR: - Put latches in between the states of the processor. These latches buffer the data and control signals as they go through the pipeline.

What if we have:

```
addi $1, $2, $3
lw $4, 0($1)
```

```
sub $1, $4, $3
```

Add will go through instruction decode. Then `lw` goes through instruction decode. It reads `$1` from register file, but the `addi` hasn't calculated `$1` yet let alone stored it back to register file!

This is a *Hazard* := any situation that prevents a processor from starting the next instruction in the next cycle. Come in 3 types:

1. Structure Hazard: A given resource is busy. Eg: multiple instructions come in, each needs the data port to fetch instructions.
2. Data Hazard: Need to wait for an old instruction to complete before we have the data we need. motivating example.
3. Control Hazard: deciding on a control action depends on the result of the previous instruction. eg: we need to wait for a `bne` to evaluate, so we know what to fetch.

Structural Hazards

Won't go into much here, because our MIPS architecture has no issues.

Consider an alternate architecture where we only had one set of memory (no segregation between data and instruction memory). Then could only fetch for one instruction at a time.

When we can't start a new instruction, we need to *stall* processor (send a no-op through pipeline, also called injecting a *bubble*). More stalls means more CPI diverges from 1.

Data Hazards

Instructions depend on completion of data access by a previous instruction.

Mind: Transparent latches! Register file can read and write to itself in the same cycle! (execute during "first half" and "second half" of cycle, think about it). So try to align dependent register file writes and reads into the same cycle.

How do we deal with this?

Cheaty approach (1): "let the programmers worry about this". In software, can require inserting independent instructions (eg: no-ops) or reordering instructions to avoid any data hazards. This wouldn't be portable, because software would have to know underlying chip's pipeline info and be compiled for that specifically. So while we do encourage compilers to separate and "reorder" statements in the higher-level language, we assume that sometimes the compiler won't avoid all data hazards. More of an optimization than a requirement – we need a hardware mechanism.

Two possible mechanisms:

Data Hazards: Stalling

Problem: correct data hasn't been written back to register file yet. Stalling solution: wait for it to be written to the register file.

Every stage can throw a *stall*, which will freeze the stage and any stage before it; all stages ahead continue.

1. Detect the hazard. Do this by comparing the read registers in the ID stage to the write registers of all later stages.
2. Stall the pipeline. For our arch, we just need to stop IF and ID (ID because it might need register values for later, IF because we don't want to drop instructions). Do this by not incrementing the PC, and not writing the IF/ID registers. Then, insert no-ops by setting all control signals that propagate onwards to zero.

Data Hazards: Forwarding

Problem: correct data hasn't been written back to register file yet. Forwarding solution: Use the data while it's still in transit to the register file.

Eg: subtract word needs the result of an addition. So:

1. Send the add through as usual. When the add hits EX, the subtract will finish ID.
2. Add is stored in EX/MEM latch, subtract is in ID/EX latch. Now, replace the subtract's register value with the on in the ID/EX latch.
3. Continue on.

This requires extra datapath connections, and a mechanism to keep track of which register is in the latches. Notate as `EX/MEM.RegisterRd` ("which register was used as `rd` in the `EX/MEM` latch. Appropriate conditions aren't too complex; mainly stuff like"If the later instruction would write to the register file `$0`, and the current instruction uses `$0` as `rs`, then we should forward.". We just have to watch out for instructions in the EX stage (would write to register file) and MEM stage (could read from memory). A mux watches these conditions, and switches the ALU inputs. If two instructions would both cause hazards, forward the *earliest* one in the pipeline (it has the most up to date value). Ie: use EX instead of MEM.

Can't always forward our way out of trouble. Sometimes, the value we need isn't later in the pipeline. Eg: we load, then add based on that load. When the add hits `ID`, the load isn't even in the MEM stage yet. This *Load-Use* hazard requires 1 cycle stall (once value is out of MEM, we forward it immediately). Processors automatically insert a bubble when the load use condition is met.

Precise conditions are in the slides, `4_10`.

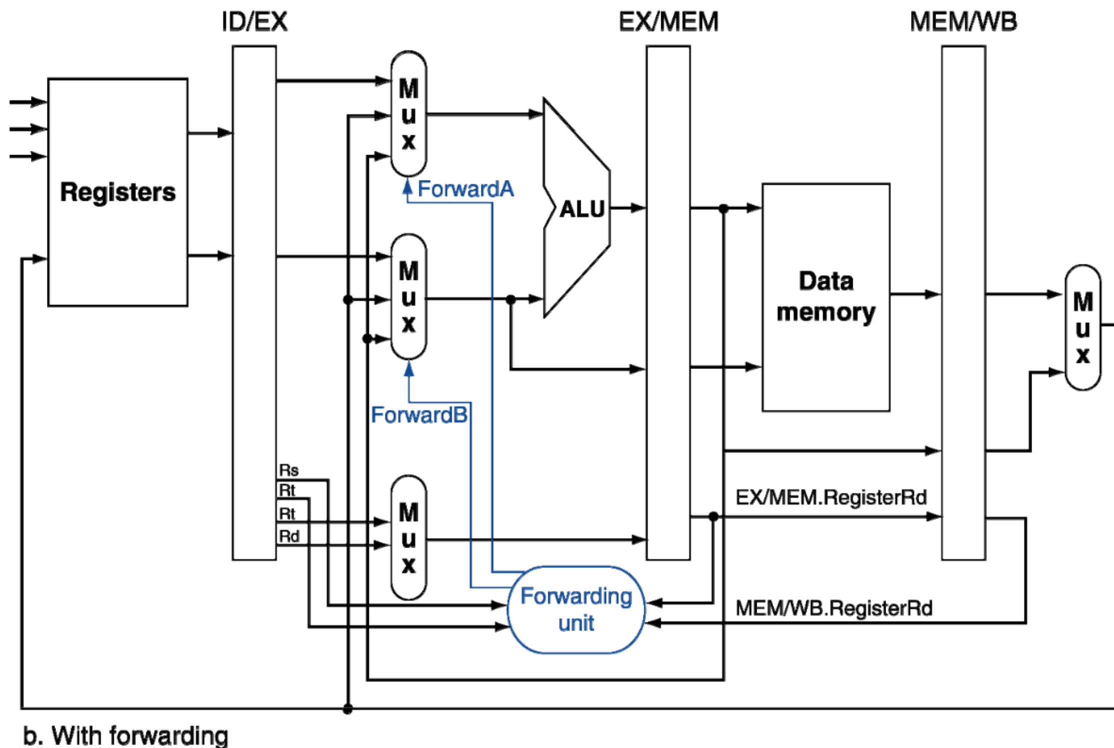

Figure 2: Data Forwarding Path

```
# Remember, forward the _latest_ value. That's why this logic is long.
MEM hazard
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
  and not (
    # This is the EX hazard condition. Don't forward
    # if it's true.
    EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs)
```

```
  )
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs)
  )
  then
    ForwardA = 01
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
  and not (
    # This is the EX hazard condition. Don't forward
    # if it's true.
    EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt)
  )
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  then
    ForwardB = 01
- 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
```

Control Hazards

Control hazard is like a data hazard that involves the PC. Need to wait for result of a `bne` before we know what to fetch (so can't pipeline the next IF).

In this class, we assume the result of the `bne` is available for forwarding after the EX stage. MIPS adds hardware to do a quick-compare of 2 registers in the ID stage itself. This means the BNE doesn't have to go through the EX (ALU) stage; only need 1 cycle stall. But that's outside of this class's model.

We can stall to avoid control hazards. We can also expose *branch delay slots*; equivalent to "let the compiler worry about control hazards". Branch delay slots are where the regular stall bubbles should be (2-3 after the branch instruction); are executed regardless of whether branch is taken.

Stall is feasible here, less so for deeper pipelines. They need to *branch predict* := make assumption about whether branch, and start pipelining instructions. If wrong assumption, transform all "wrong" pipelined instructions into bubbles. We'll predict too – even though it doesn't help us much here.

- Static Branch Prediction: based on typical behavior (eg: backwards branch indicates loop).
- Dynamic Branch Prediction: hardware counter for every branch instruction. Use heuristics to predict what will happen next (eg: "it was taken before"). Routinely reach hit rates of 0.96 nowadays.
  - 1-bit branch predictor: store 1 bit for each branch, remembers if it was taken before. Do that.
  - 2-bit: same, but requires 2 successive misses to change its prediction.

Exceptions

Suppose the ALU overflows and we need to trigger an exception. This forces the PC to jump to a certain address based on the type of exception, and sets the "cause register" to indicate more details about processor error. Old PC is set to `EPC`, so we can use that to diagnose error in OS (and to jump back eventually).

These make pipelining complicated. Look through the slides.

Going Superscalar

you're not dealing with the average MIPS processor anymore, FRIEZAAAA

Can go superscalar (CPI $>$ 1). Advanced pipelining section.

Going Superscalar: Multiple Issue

Previously, we were executing instructions in parallel via pipelining. Instructions in different stages at same time. Now, we "widen" the pipeline by duplicating functional units. Multiple instructions in same stage (eg: replicated ALU). Called *multiple issue*. 2 approaches to avoiding dependency problems:

1. *Static Multiple Issue*: Compiler does scheduling and hazard avoidance by inserting no-ops. Explicitly groups instructions into "issue slots", which ISA issues together.
2. *Dynamic Multiple Issue*: Processor reads instructions as stream, automatically chooses which instructions to issue each cycle. Needs

Going Superscalar: Multiple Issue: Static Multiple Issue

Static: compiler groups instructions into *issue packets*; a group of instructions that can be issued in a single cycle (eg: processor allows one add and one load per cycle. Why different types? Cheaper. Eg: loads require just an adder instead of an ALU). Think of an "issue packet" as a very long 64 bit instruction (sometimes called *Very Long Instruction Word, VLIW*).

Compiler needs to understand architecture and remove hazards (Eg: no dependencies allowed inside a single packet). Reorder instructions to maximize parallelism; if compiler can't find good ordering, it can:

1. Shoot out some no-ops.
2. Modify the code (change offsets) to remove data dependencies.
3. Unroll loops to expose parallelism. In different loop runs, use different sets of registers (when register renaming, maximize distance between consecutive uses) to reduce data dependency.

How to loop unroll:

1. Replicate the instructions n times.
2. Look for quick optimizations (eg: instead of decrementing `i` n times, just subtract n at the end of the unrolled loop). You can often delete instructions altogether.
3. If the loop iterations are independent, make them use different registers (register renaming).

Remember: if you reorder an add, adjust all later offsets that depend on that quantity! Eg: `i += 1` can be done early, but subtract 1 when indexing into the array later (via immediate `-4(t0)`).

Going Superscalar: Multiple Issue: Dynamic Multiple Issue

Put the superscalar logic in the CPU instead of the compiler. CPU dynamically decides how many instructions to execute per cycle. Can reorder independent instructions. Very complicated logic.

Going Superscalar: Speculation

"Guess" what to do with an instruction, start operating as soon as possible. Roll back CPU state if guess is wrong. Much more complex than branch prediction, because computer state is actually affected (eg: writes to memory!).

What if exception during speculation? Static speculation adds support for deferred exceptions to ISA, dynamic speculation buffers them transparently.

This can improve CPI for multiple issue processors, even when no more optimal scheduling exists.

Going Superscalar: Multiple Cores

Multiple issue had problems. Even with optimal scheduling, most programs used a lot of no-ops. Just not enough parallelism. Pointer aliasing causes dependencies, etc. Other issues (memory delays, bandwidth issues) meant that the industry stopped chasing superscalar processors. Instead, they moved towards multi-core processors.

Chapter 5: The Memory Hierarchy

On to chapter 5.

Section 5.1

- SRAM > DRAM > flash > magnetic disk. SRAM is stupidly expensive.
- "Block" is synonymous to "line" in terms of caches (both are "unit of data copy").

Apparently people are trying to use flash in the role of DRAM. Is the durability that good now, or is it a different technology?

Section 5.2

Problems: How do we manage cache? How do we know where data is in cache?

Solution 1 is a *Direct Mapped Cache*. Lowest overhead, but not smart. Address uniquely determines location in the cache; use the lower `n` bytes to figure out which cache block it's in.

Associativity := how many cache blocks a memory address could possibly reside in. Here, it's 1 – because a given address can only sit in one block (the one based on lower order bits). Like a hash table without a linked list as a bucket.

Say we're trying to find an address.

```
0000 1101 1111
_a__ _b__ _c__
```

- `b`, the *index*, tells us which cache line to search for the entry in. It acts like the key to a hash table.
- `c` is the *byte offset*. Our cache doesn't store just one byte; it stores *blocks* (or *lines*) of a certain size. We just need to pull in the appropriate cache line, so we discard the `c` bit range. `c` compensates for memory being addressed in bytes.
- `a` is the entry's *tag*. A given cache block (hash bucket) could store multiple possible addresses, so we store the top (nonhashed) bits of the address and disambiguate.

Section 5.3

Large block size can be useful. But if cache size is fixed, it would also imply fewer blocks. Larger blocks have larger miss penalties too. Mind these tradeoffs.

When the cache misses, the CPU stalls the pipeline until access finishes.

Reads are straightforward. Writes are hard, because they make cache and memory inconsistent. Several options when writing a value.

If the write hits:

1. Write-Through: keeping memory and cache in sync is hard, so just update memory whenever you write to cache. Writes will take a long time. TLDR: Write at write time.
2. Write-Back: only write to cache, and mark block as "dirty". When it's time for that block to be evicted, *then* write to memory. Cache replacements take a long time. TLDR: Write at eviction time.

Can ameliorate delays in both cases by using an asynchronous write buffer (output queue for pending writes).

If the write misses:

1. If you chose write through:
   - Allocate on miss: fetch the block into cache, then write to it as usual.
   - Write around: don't fetch it into cache, just write to memory directly. Why? Read locality and write locality may differ.
2. If you chose write back: you don't have the above option, because you're keeping the entry in cache anyways.

Update our performance equations, add the cycles spent stalling for memory.

$$\text{Memory Stall Cycles} = \frac{\text{Memory Accesses}}{\text{Program}} * \text{Miss Rate} * \text{Miss Penalty} = \frac{\text{Memory Accesses}}{\text{Program}} * \frac{\text{Misses}}{\text{Instruction}} * \text{Miss Penalty}$$

We also have *Average Memory Access time (AMAT)*. Takes average hit time into account too.

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

Section 5.4: Associative Caches

Review the slides. TLDR:

- n-way set associative := each cache set (line) contains n entries (like linked list in hash table bucket).

Section 5.5: Multilevel Caches

TLDR: Equation for performance and CPI changes.

Definitions:

- TCPI := Total CPI
- BCPI := Base CPI
- PCPI := Peak CPI. 1 for single-cycle or pipelined.
- DHCPI := Data Hazard CPI.
- CHCPI := Control Hazard CPI.

Equations:

- TCPI = BCPI + MCPI
- BCPI = PCPI + DHCPI + CHCPI
- MCPI = miss % * (L2 latency + L2 miss % * L2 Miss Penalty hdots)
- DHCPI = % Relevant Instructions * % Hazard Rate * Hazard penalty

Don't forget to count the relevant instruction frequencies. Also don't forget; if CT changes, these penalties may change (they're in cycles).

Section 5.6: Virtual Memory

We already have a strong foundation with this.

TODO: DO THIS

Chapter 7: Multicores, Multiprocessors, and Clusters

Strong Scaling.

Multithreading means a different thing in hw land. It's like multicore, but the ALU and functional units are shared. switch to different thread in stall. can also do simultaneous multi threading (intel hyperthread), which is same thing but in parallel.

TODO: DO THIS, Snoopy etc

FINAL REVIEW

DEFINITIONAL MISUNDERSTANDING!: - Compulsory Miss := "We have never had this data in cache before". - Conflict Miss := "We used to have this in cache, now we don't". Requires us to keep track of all history, not just looking at the cache entry we replaced!

When we say a TLB has "64 sets and is 8 way set associative", we mean that there are 64 hash buckets, and each bucket holds 8-length linked list.

TODO: "Dependent instructions cannot be placed together". What does that mean? To what extent? Eg: our solution to 5.