Note 13: RISC-V Datapath

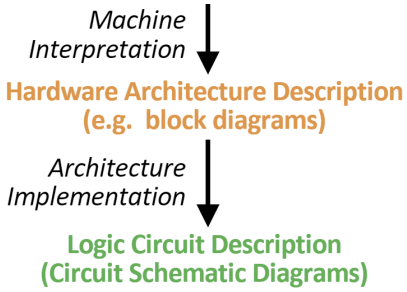
Kevin Moy

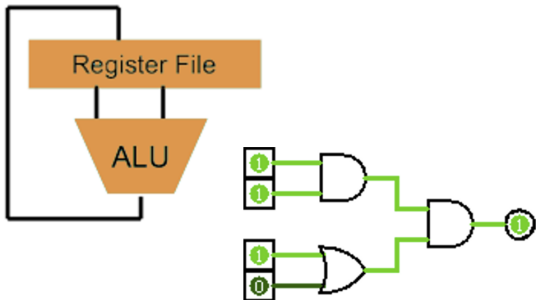
Abstract

In this note, we'll going to discuss the RISC-V CPU datapath- essentially the physical means of actually executing instructions. This is a long note- please bear with it! It'll be essential in knowing how to build your CPU later on.

1 Introduction

Now processing pure binary instructions, we are now at the lowest level of representation and interpretation in computer architecture, as shown by this diagram:





We are now at the most physical stage, where hardware will execute what the binary wants. At this point, we break the abstraction barrier between software and hardware and need to connect them in such a way that hardware *responds* to software instructions.

High-level languages become machine language (binary) through compilation and assembly and linking, and we now **merge** it with the physical aspects

of hardware: registers, clock circuits, gates, etc. We are, in essence, bringing together the architect and the construction workers.

Our goal overall is to make a **CPU** (central processing unit), the circuit that executes instructions we feed it. Given some assembly instruction, which always has some ML (binary) representation, the CPU will perform the action that the instruction describes, and have its physical state updated to reflect the instruction execution: for example, if my instruction is `li t0 6`, register `t0` will have value 6 after execution.

2 The CPU

The CPU is the central processing unit: it processes stuff and is the central unit in your computer that does so. The CPU generally consists of of two main parts: the datapath and the control(ler):

- **Datapath**: Hardware that responds to instruction and executes it physically (construction worker). This is essentially the logic that *reacts* to the control part of the CPU and does its bidding.
- **Control**: Determines and allocates role for each datapath component based on instruction (the architect). This is the brains of the CPU. For example, the control portion tells the datapath if we generate immediates or not, writing back to `rd` or not, etc.

Let's take an example instruction: `addi t0 x0 6`. The datapath is the part that actually "reads" this instruction from memory, and send it to control logic (it doesn't have any idea what the instruction does). The control logic will then analyze the instruction, like finding out what instruction it is, what `rs1/rs2/rd/imm` are and if they are needed, as well as what appropriate **control signals** to activate for datapath components.

In this case, the control logic tells the datapath the operands are registers `x0` and immediate `6`, then tells it to perform the `add` operation on those operands (through the ALU). Next, the control tells the datapath to store the adder output into destination register `t0`. Finally, it increments `PC` in datapath by some amount (4 or some other offset).

Sure, so the control logic is the brain. How do we start implementing a brain? In reality, the control logic is not an actual brain but is configured in such a

way that out of all valid instructions, it will know what to analyze and tell the datapath the right settings correctly.

So that leads us to a better question: **what operations should our datapath support?**

Even with reduced instruction sets, there's still a lot to implement: 6 formats of instructions, specifically, for base RISC-V. Recall that that each format, conceptually, does a different thing:

- R-Type: Arithmetic instructions
- I-Type: Immediate instructions, includes load and `jalr`
- S-Type: Store instructions
- SB-Type: Branch instructions
- U-Type: Upper immediate instructions
- UJ-Type: `jal` (jump and link).

But even though these formats have different behavior, RISC-V has constructed them in a way such that their instruction fields are as similar as possible. So thankfully, we won't need a separate circuits for each of these formats, but instead just small modifications to our original single circuit to account for all 6.

Each type has specific functionalities that we want isolated from and unaffected by other instructions. It's now time to build our CPU. Starting from the basics, we take a single format then we're going to implement, then slowly build off this to account for all six formats, and eventually, the entire RISC-V datapath. We must ensure that as we build to account for new formats, we still are able to fully implement old ones.

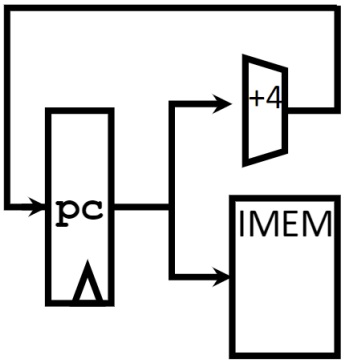
3 Building R-Type Datapath

Let's start with R-Type instructions. They are generally the easiest instructions to execute, and implementing them gives the most solid base for implementing the other five instruction formats.

What actually needs to happen for an R type instruction to execute? Again, let's consider an example: `add t0 t2 t3`.

The first step is actually processing the instruction- in other words, knowing that we're executing an `add` instruction. Next, we need to extract relevant fields from the instruction: registers `rd=t0, rs1=t2, rs2=t3` as well as operation fields `opcode`, `funct3`, and `funct7` that specify the exact instruction (`add`) to execute. Next, we evaluate our operands, in this case `rs1=t2` and `rs2=t3` and getting the values stored in those registers. Then, we perform the `add`, and finally store the result back into `rd`. The Verilog instructions on the green sheet give a succint, code-formalized description of this process for each instruction. At the very end, we must increment `PC` by the correct amount: for R-type instructions, this amount is always 4.

Remember that `PC` holds the address of the current instruction in memory. So it makes sense that we need something to fetch this instruction out of memory. Consider this diagram below so far:



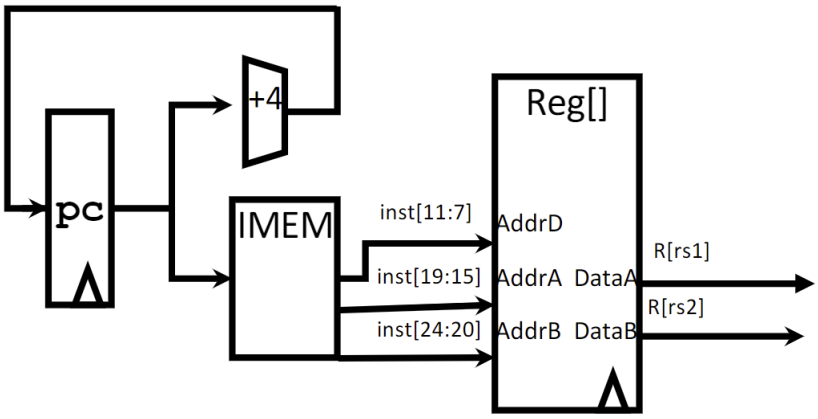
On a clock tick, the PC register sends its value to `IMEM`. `IMEM` then uses it to fetch the current instruction, since PC's value was an address. It will output the (32) instruction bits.

Note we also add 4 to `PC` and write it back to `PC`, as a means of updating it to hold the address of the next instruction. On the next clock tick, the new instruction executes.

So IMEM will parse the instruction into the different fields, and output different 5-bit fields of the instruction, which actually specify which registers to use! These bits will go into our **register file**, a new piece of hardware

that basically stores all our registers (with their values). Note that `RegFile` doesn't include `PC`: `PC` is a special register only determined by certain parts of the datapath and certain instructions. We certainly don't want to access it with an R-type instruction.

So now we have this updated datapath below:



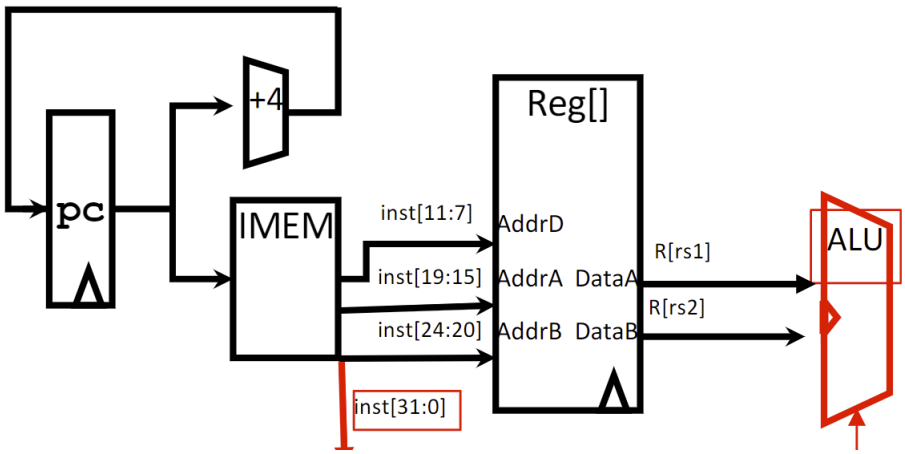
In `RegFile`, or `Reg[]` in the diagram, we actually only have 31 registers. Why 31 and not 32? 0 is hardcoded, so we don't really need to make a register for `x0`.

Let's analyze `RegFile` more closely. As inputs, we have the **addresses** for `rs1` in `AddrA`, address of `rs2` through `AddrB`, and address of `rd` through `AddrD`. As outputs, we have two ports that allow us to extract the values of `rs1` (`DataA`) and `rs2` (`DataB`).

We also actually have another input port, `portW`, which will contain the value we want to *write back* to the destination register specified. We have a control signal `RegWEn`, for Register WriteEnable, that will allow writing into registers. So for R-type instructions, `RegWEn` is true, but when do we want it to be false? Well, for instructions that don't write to registers- branch instructions!

Next we need to add the **ALU**, or the arithmetic logic unit. This is the easiest to understand: it is simply the block that does math. The `ALUSel` control signal tells us which operation the ALU should perform on its inputs.

So our final R-Type datapath:



So now we can support a large number of R-type instructions: not only `add` but also `sub`, `slt`, `or`, `and`, etc. The specific R-type operation we want to perform will be specified by `funct3` and `funct7`.

Finally, we have to write our result back to `rd`, our destination register, in `RegFile`. And we are done.

4 Building (Arithmetic) I-Type Datapath

Now let's move on to building a datapath for arithmetic I-type instructions. As always, we take an example: `addi x2 x1 -50`.

So as usual, inside our control logic we parse the instruction and send the appropriate fields to the datapath. In this case, `rd=x2`, `rs1=x1`, and `imm=-50`, and this will be reflected in the bits set to the datapath.

Remember that `opcode` tells us format, and `funct3/funct7` tells us the specific instruction.

For the most part, implementing R-type has already done most of the work for us! We know how to add as well as read and write register values. However, we haven't implemented logic for our immediate yet. We'll do that here!

For I-type instructions, the instruction fields differ a bit from R-type. Instead of `rs2`, there's `imm`. So we account for changes in that, when we detect an I-type instruction (`opcode` based!).

The first thing is that bits 20 to 31 of the instruction will be fed to the **immediate generator**: like the ALU, another abstraction that outputs the (32-bit sign extended) immediate we want based on those 12 bits as well as a control signal `ImmSel`. `ImmSel` will differ based on whether we want I-type immediates, or S-type, or some other immediate.

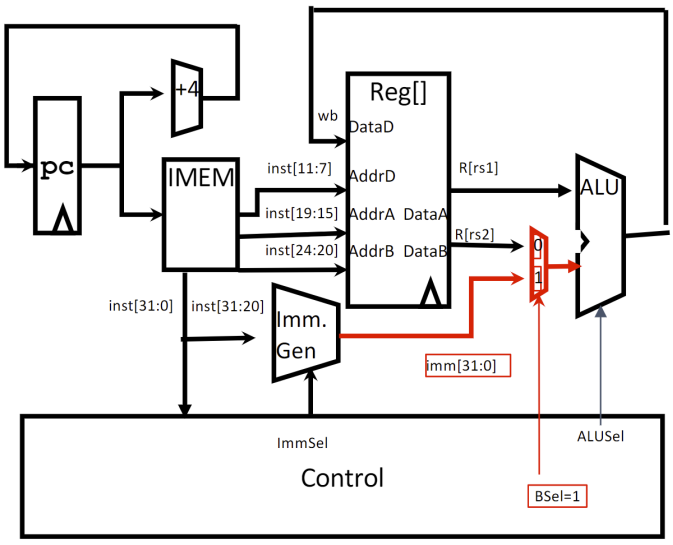
How can we decode the immediate specified by the instruction, i.e. get the immediate we want? Well since we use 32-bit values but only have 12 bits specified in `imm`, we need to sign-extend. That's it, and that will be the only job of `ImmGen` as far as I-type instructions go.

We also have to set `ImmSel` to select I-type and then output our immediate.

However, note that for I-types we don't actually want to read `rs2` into our ALU anymore, we want to read our `ImmGen` output. So it makes sense to have a MUX to select the immediate over `rs2`'s value here. We call this control signal `BSel`, and set it to 1 to indicate getting the immediate (`BSel=0` for R-type instructions).

And the rest of the process is the exact same as R-type instructions: feed into ALU and write ALU Output to `rd`, then increment `PC` by 4. So we don't need to do anything else, and we've now implemented an I-type datapath!

Here's our updated datapath now:



5 Building (Load) I-Type Datapath

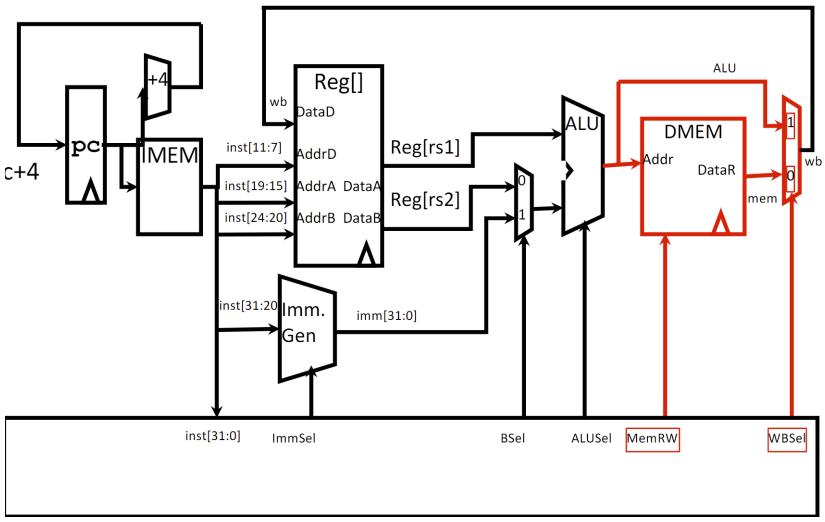
Next up we have the second type of I-type instructions: loads. Wait, haven't we already implemented I-types? Certainly. But there's something different about load instructions that we haven't covered yet in our datapath...

...

Oh, yeah, we need to now implement memory!

Notice that for load (and store) instructions, we are actually adding the register address and the immediate and using the sum as an index into memory. We still have `rd` to write back to, so the instruction isn't totally different, thankfully.

So really, we want to implement `lw` in our datapath. Thus we add the **data memory component**: very similar to instruction memory except for now we actually can fetch arbitrarily *from* data memory. We'll call this data memory component `DMEM`.



Notice the data memory block takes yet another control signal `memRW`, which essentially indicates if we want to read (load) or write (store) data to `DMEM`. The `DMEM` block also takes an address from the output of ALU (register address + offset) and outputs a 32-bit value `DataR`.

Notice we also added another MUX to choose between `DMEM` output and

ALU output, controlled by *another* control signal `WBSel`, which selects which output (ALU or DMEM) to write back to a specified `rd`. We'll call this the writeback select MUX.

Ideally, our data memory has one input port, for data in, and one output port, for data out. If the write-enable control signal is set to 1, we write data in to our address, and if it's 0 we basically read the data at the address and output it (as a 32 bit value).

In general, instructions certainly don't need to utilize every single datapath component. In fact, most these components are unused. For example, for R-Type and I-type instructions, `DMEM` is completely unused. However, load instructions require EVERY component of our datapath! **The load instruction exercises our critical path, and thus determines how fast our clock cycle can be.**

Finally, let's say to support ALL load operations (`lb`, `lhu`, etc.) This is where `funct3` comes in: it indicates the size to load AS WELL AS whether the load is unsigned or not (MSB=1, then unsigned). But we'll assume the DMEM handles all this, as far as this class goes.

6 Building S-Type Datapath

For the most part, actually, our datapath is done! We just need some small additions to finish up the remaining 4 instruction formats.

So for S-type instructions, or store instructions, how do we change datapath/control signals to implement that?

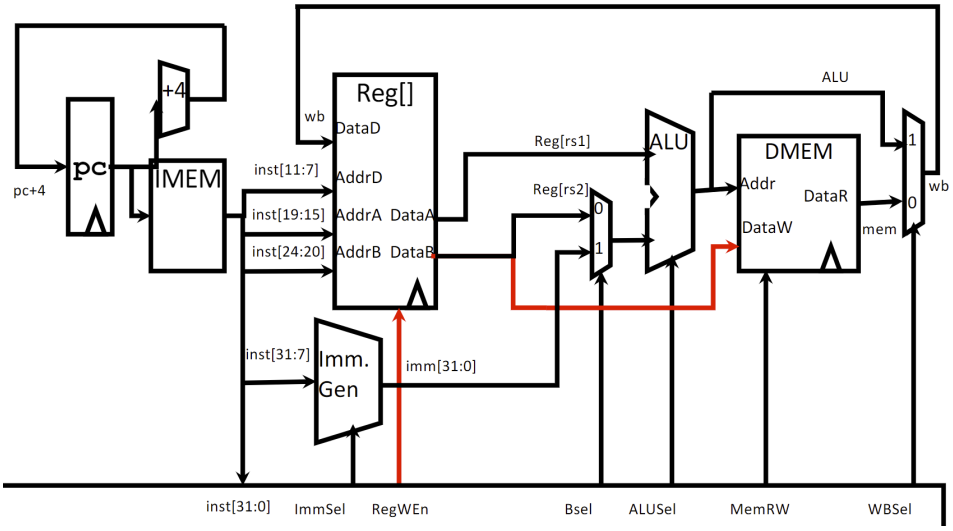
We talked about control bits or control signals: they instruct our datapath/hardware how to deal with different instructions.

We can fetch instructions from memory. We have a mechanism for decoding instructions, and can easily just change `ImmSel` to signal S-type. We need to fix the immediate generator to account for S-type instructions. We set the `ALUSelect` to `add`. We can now access memory correctly.

The key fix is we need to now set `MemRW` to 1 (writing to memory) and also pass `R[rs2]` to `DMEM`, since the value in that register is what we are writing.

We also don't write to `rd`, so we want to make sure that path is avoided during S-type instructions.

Let's implement the wires we now need:



We have added a wire that bypasses BSel and connects `R[rs2]` directly to the `DMEM` input for writing. We also added the `RegWEn` control signal which indicates if we want to write back something into `rd` or not. In the case of S-type, we don't have an `rd`, so we set that to 0 in this case.

7 Building SB-Type Datapath

Next up is SB-type instructions, or branching instructions.

So far, our instructions have been about reading and writing registers or memory. But what about branch instructions- instructions that change the flow of execution? These instructions actually modify the `PC` register- either to `PC+4` OR to some label OR to some register containing an address. Branching instructions are either **PC-relative**, i.e. we need calculate `PC+imm`, or they are **absolute**, which means PC is actually set to some pre-stored (in a register) address.

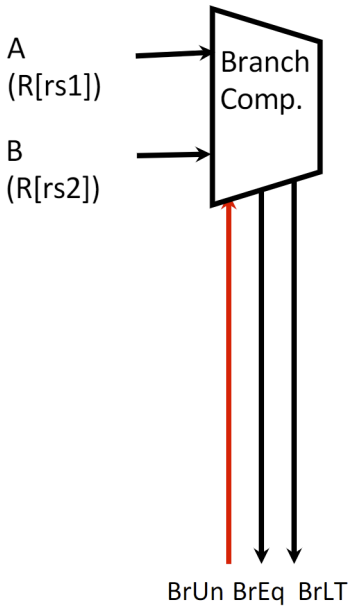
The opcode and funct3 unique determine the specific branch instruction. `rs1` and `rs2` determine the comparison registers. Finally, this annoying ass

jumbled up `imm` field.

Branching involves two different operations. The first is comparison: checking if `rs1=rs2`, or if the generic comparison between `rs1` and `rs2` returns true. The second is `PC = PC + imm`. Unfortunately, the ALU can only do one of these things- not both.

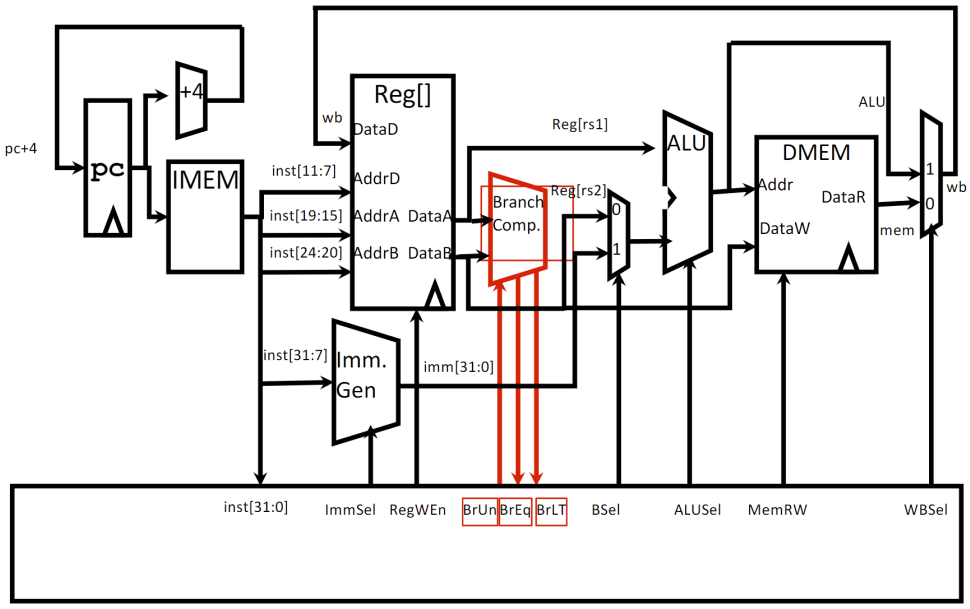
So to handle both, we add a **branch comparator**: it takes `rs1` and `rs2` and a control signal `BrUn`: if `BrUn=1`, unsigned comparison for `BrLT`, and signed `BrUn=0`. Then, it also OUTPUTS two control signals: if `rs1=rs2`, then `BrEq=1`. Same with `BrLT` (we don't need `BGE` because we can just negate `BrLT`).

Below is a diagram of the branch comparator as a black box for now:



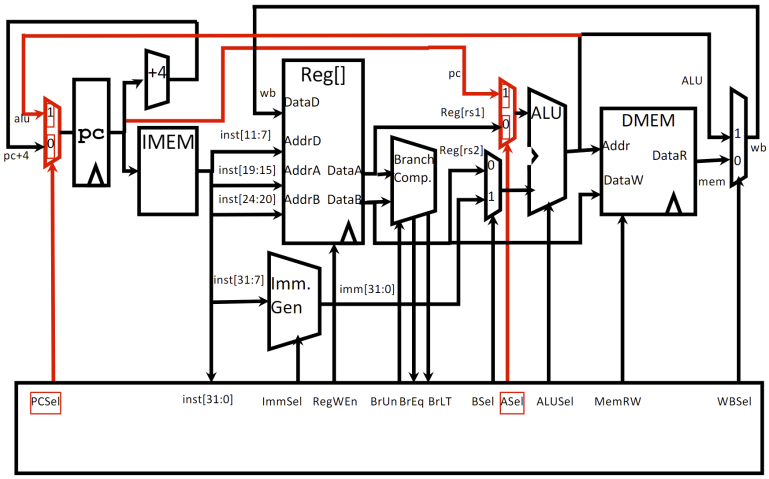
So if any of these output control signals are true, then we take the branch and must accordingly change `PC`.

Adding it to our datapath, we now have:



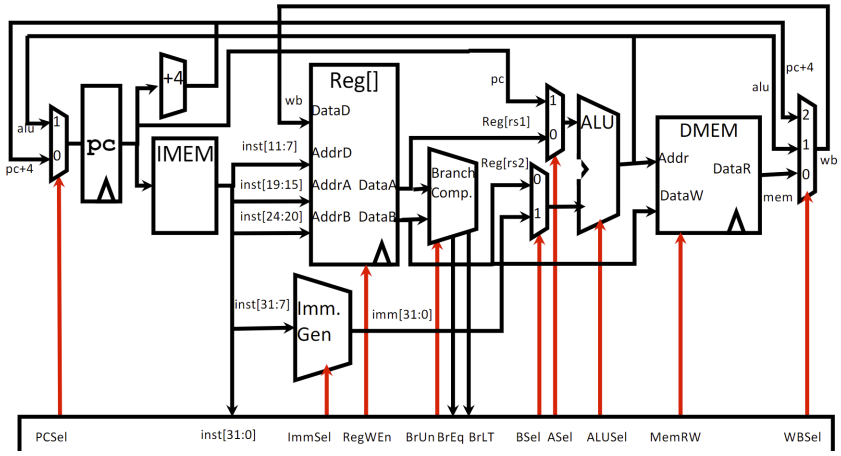
See how it takes in the two register values `R[rs1]` and `R[rs2]` as well as control signal `BrUn`. It'll output some control signals as well, `BrEq` if `beq` or `BrLT` if `bge` or `blt`.

One last thing: we need to choose between `PC=PC+4` or `PC=PC+imm` (branches/jumps). So we add one more MUX, now with the `ASel` control signal, to choose between `PC` and `R[rs1]`:



Why `R[rs1]`? This is for adding an immediate, calculated already by `ImmGen` and propagated by the `BSel` MUX, to a register containing an address. For PC-relative addressing, we calculate `PC+imm` in the ALU which we propagate to the write-back select MUX.

But we need ONE MORE MUX still: to select between `PC+imm`, an output of the ALU, and `PC+4`. We reflect this below with another control signal `PCSel`, which we only set to 1 in branch (and jump) instructions:



8 Building (Jump) I-Type Datapath

Take a break right now if you haven't already. This been a long note.

So our Datapath we've implemented so far looks phat. Yet we still have two instruction formats left to implement. We can't possibly need too many other modifications, right???

Right???

Thankfully, that's pretty much correct.

We want to implement jumping I-type instructions- specifically, `jalr`. We already implemented I-types, so we already know how to interpret and decode a `jalr` instruction. However, because it's a jump, just two small changes need to be made.

The first is write `PC+4` to `rd`, so we need to connect that to `RegFile`. Second, we set `PC = R[rs1] + imm`- but we already implemented this so we just need to change control logic. That's it!

But we already connected `PC+4` to the write back select MUX! So now `WBSel` must become TWO bits: `WBSel=2` if `jalr`, `WBSel=1` if standard ALU output is to be written to `rd` (R-Type or other I-type instructions), and `WBSel=0` if we want to load into registers from data memory.

9 Implementing J-Type Instructions

Finally, J-type instructions.

This is for the `jal` instruction: yet another fucked up `imm` field. Remember `jal` saves `PC+4` into `rd` (`ra` by default). We implemented this with implementing `jalr`. We also calculate an offset immediate and set `PC = PC + imm`. Again, already implemented with branch instructions.

So we don't need to actually do anything!! Thank fuck for that.

Remember that `jal` can support $\pm 2^{19}$ location jumps, 2 bytes apart, or $\pm 2^{18}$ for standard word instructions.

NOTE: Implementing U-type instructions is just like I-type, except we need an upper immediate, so all changes go into `ImmGen`.

We're Done!!!!!!!!! Congrats on making it through this note!!!