Understanding the `control` Module in a Simple CPU

What Is the `control` Module?

In any CPU, the **control unit** is like the "orchestra conductor"—it interprets the instruction that the CPU reads from memory and tells the various parts of the CPU what to do for that instruction. The specific control module here is designed for a simple computer processor using a RISC-V architecture, but the ideas apply to many basic CPUs.

Inputs to the Module

- **opcode**: Tells roughly what kind of instruction this is (like "add", "load from memory", "branch," etc.).
- **funct3, funct7**: Give finer details about which specific operation to perform, mainly used for arithmetic instructions.

Outputs from the Module

The module generates several **control signals** that direct the processor's components:

Signal	What It Controls
alusrc	Chooses whether an arithmetic operation uses a register or an immediate value (a number provided directly in the instruction) as its second input.
regwrite	Turns on when the CPU should write a result to a register.
memwrite	Turns on when the CPU should write data to memory.
memread	Turns on when the CPU should read data from memory.
branch	Tells the CPU whether this instruction is a conditional jump ("if..." statements in code).
aluop	Tells the "math unit" (ALU) what operation to perform: add, subtract, and, or, etc. (encoded as three bits so you can have up to 8 operations).

How Does It Work?

When the CPU reads an instruction from memory, the **opcode** and other instruction bits are sent to this control module. Here's a simple analogy:

- The opcode says "what kind of work" the instruction wants:
    - **Arithmetic** (add, subtract, etc.): Use the registers and the ALU.
    - **Load/Store** (move data between memory and registers): Use data memory.
    - **Branch/Jump** (change the flow of the program): Adjust the program counter.

For each instruction type, the module decides:

- Should the ALU (math unit) get its numbers from two registers, or one register and an immediate value?
- Should the ALU perform an addition, a subtraction, or something else?
- Should the result of the ALU go to a register, or be ignored?
- Should data memory be read or written?
- Is this instruction a branch (jump)?

The actual module is implemented with a **case statement** that checks the opcode and, for arithmetic instructions, further checks funct3 and funct7 to distinguish between types (like ADD vs. SUBTRACT).

An Example Table

Here's how the logic changes for different instructions:

Instruction Type	ALU Uses Immediate?	Writes Register?	Reads Memory?	Writes Memory?	Branch ?	ALU Operation
Add, Subtract	No	Yes	No	No	No	Add or Subtract
Add Immediate	Yes	Yes	No	No	No	Add

Load (from mem)	Yes	Yes	Yes	No	No	Add (address calculation)
Store (to mem)	Yes	No	No	Yes	No	Add (address calculation)
Branch Equal	No	No	No	No	Yes	Subtract (comparison)
Jump	-	Yes	No	No	Yes	Add

Why Is This Needed?

Every time an instruction is run, the processor's parts must know their jobs: where to get data, what math to do, where to send the results, and whether to change the program sequence. The control module looks at the instruction and sets all those "switches" correctly, many times per second, so that the CPU does the right thing.

In Summary

- The control module acts as the **decoder and director**:
    - It translates an instruction into specific signals that tell the rest of the processor what actions to perform.
- This is a universal idea in computer architecture, making it possible for a handful of logic circuits to perform all the tasks software calls for.

Put here for understanding purposes!

`control` Module – Explanation

Inputs & Outputs

Signal	Width	Direction	Description
`opcode`	7	Input	Instruction[6:0]; primary opcode for instruction type

`funct3`	3	Input	Instruction[14:12]; further distinguishes operation (ALU)
`funct7`	7	Input	Instruction[31:25]; further distinguishes R-type ops

Name	Width	Output	Meaning
`alusrc`	1	Output	1: ALU input is immediate; 0: ALU input is register
`regwrite`	1	Output	Write enable for registers
`memwrite`	1	Output	Write enable for data memory
`memread`	1	Output	Read enable for data memory
`branch`	1	Output	Branch/jump indication
`aluop`	3	Output	Select ALU operation (`ADD`, `SUB`, `AND`, `OR`, etc.)

Working

Default Control Signals

At the start of every combinational evaluation, all control signals are defaulted to safe/inactive values:

verilog

```
alusrc = 0; regwrite = 0; memwrite = 0; memread = 0; branch = 0; aluop = 3'b000;
```

This prevents accidental propagation of stale or incorrect signals.

Instruction Decoding (`case` on Opcode)

The `opcode` field is the primary instruction type discriminator:

❖ R-type Instructions (`opcode == 0110011`)

- Pure register-register operations.
- `regwrite = 1; alusrc = 0; memread = 0; memwrite = 0;`
- The specific ALU operation is determined by `{funct7, funct3}`:
  - ADD (`0000000_000`): `aluop = 000`
  - SUB (`0100000_000`): `aluop = 001`
  - AND (`0000000_111`): `aluop = 010`
  - OR (`0000000_110`): `aluop = 011`
  - Default: `aluop = 000`

❖ I-type Arithmetic/Immediate (`opcode == 0010011` for ADDI, etc.)

- Register-immediate ALU operations.
- `alusrc = 1; regwrite = 1; aluop = 000` (usually ADDI)
- LUI/AUIPC are treated as ADDI-style for core simplicity.

❖ Loads (LW: `opcode == 0000011`)

- `alusrc = 1; regwrite = 1; memread = 1; aluop = 000 (ADD)`
- ALU computes the memory address.

❖ Stores (SW: `opcode == 0100011`)

- `alusrc = 1; memwrite = 1; aluop = 000 (ADD)`
- ALU computes address; result not written to reg file.

❖ Branches (BEQ/BNE: `opcode == 1100011`)

- `branch = 1; aluop = 001 (SUB)` (ALU computes difference for comparison)
- Branch taken handled by core logic outside.

❖ Jumps (JAL: `opcode == 1101111`)

- `regwrite = 1; branch = 1; aluop = 000`
- Write PC+4 to destination (link back), branch to target.

❖ Default

- Do nothing for unrecognized opcodes.

Instruction Type	Opcode (binary)	Example Instruction
R-type arithmetic	0110011	ADD, SUB, AND
I-type arithmetic immediate	0010011	ADDI, ANDI
Load from memory	0000011	LW, LH, LB
Store to memory	0100011	SW, SH, SB
Conditional branch	1100011	BEQ, BNE
Jump and link	1101111	JAL

ALU Operation Encoding

- `aluop = 3'b000`: ADD/ADDI
- `aluop = 3'b001`: SUB/branch comparison
- `aluop = 3'b010`: AND
- `aluop = 3'b011`: OR
- (Extendable for more ops as needed)

Role in the Pipeline

- **Instruction Decode Stage**: This module takes instruction bits from the IF/ID register and outputs the right control signals.
- **Datapath Direction**: Its signals select ALU operands, enable memory access, control register writes, and identify branches.

Summary Table

Opcode	Type	alusrc	regwrite	memwrite	memread	branch	aluop
0110011	R-type	0	1	0	0	0	by funct
0010011	ADDI/I-type	1	1	0	0	0	000
0000011	LW	1	1	0	1	0	000
0100011	SW	1	0	1	0	0	000
1100011	BEQ/BNE	0	0	0	0	1	001
1101111	JAL	0	1	0	0	1	000

In Summary

- **The `control` module is the "brain" for your pipeline's datapath**: it decodes instruction types and sets up the rest of the pipeline for correct computation, memory accesses, branching, and register updates.
- It supports key RV32I core instructions and can be extended for more.
- Clean separation of combinational decoding (always_comb), no latches or clocks.
- Ensures only supported instructions generate effects; everything else is safely ignored.