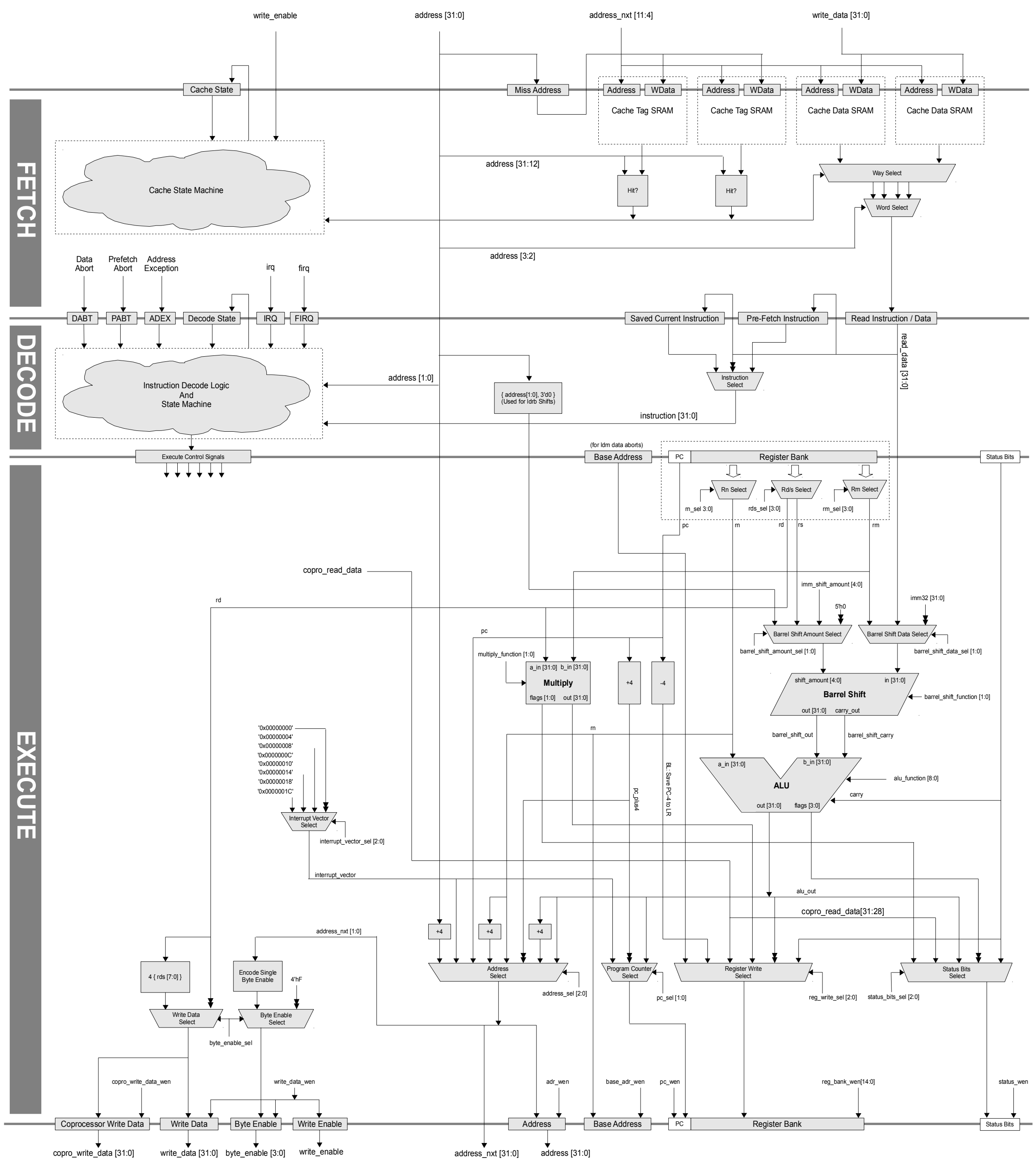
2 Amber 23 Pipeline Architecture

The Amber 2 core has a 3-stage pipeline architecture. The best way to think of the pipeline structure is of a circle. There is no start or end point. The output from each stage is registered and fed into the next stage. The three stages are;

- **Fetch** – The cache tag and data RAMs receive an unregistered version of the address output by the execution stage. The registered version of the address is compared to the tag RAM outputs one cycle later to decide if the cache hits or misses. If the cache misses, then the pipeline is stalled while the instruction is fetched from either boot memory or main memory via the Wishbone bus. The cache always does 4-word reads so a complete cache line gets filled. In the case of a cache hit, the output from the cache data RAM goes to the decode stage. This can either be an instruction or data word.
- **Decode** - The instruction is received from the fetch stage and registered. One cycle later it is decoded and the datapath control signals prepared for the next cycle. This stage contains a state machine that handles multi-cycle instructions and interrupts.
- **Execute –** The control signals from the decode stage are registered and passed into the execute stage, along with any read data from the fetch stage. The operands are read from the register bank, shifted, combined in the ALU and the result written back. The next address for the fetch stage is generated.

The following diagram shows the datapath through the three stages in detail. This diagram closely corresponds to the Verilog implementation. Some details, like the wishbone interface and coprocessor #15 have been left out so as not to overload the diagram completely.

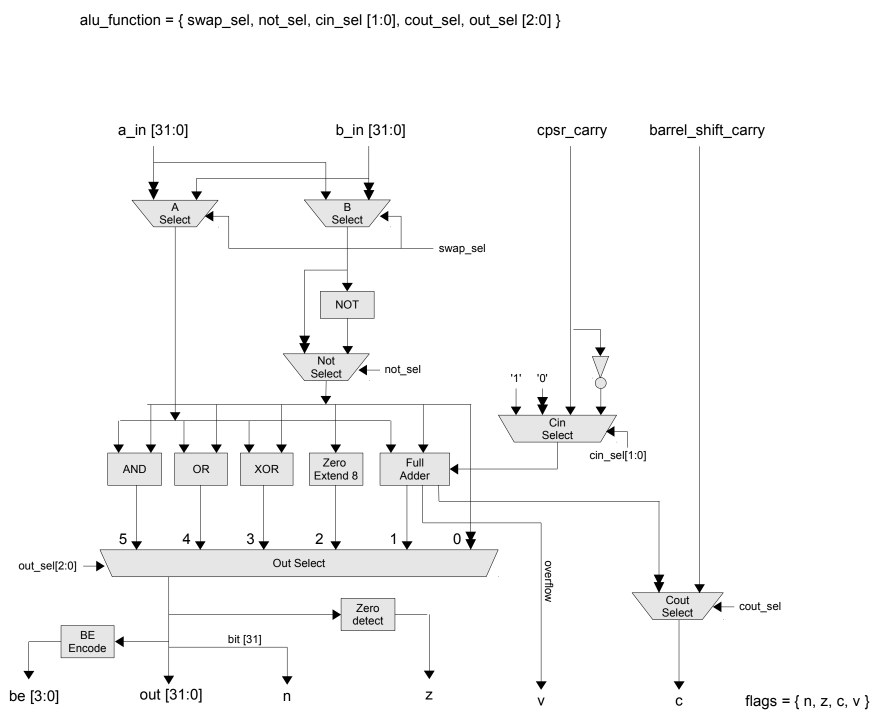
***Figure 3 -** Detailed 3-Stage Pipeline Structure*



2.1 ALU

The diagram below shows the structure of the Arithmetic Logic Unit (ALU). It consists of a set of different logical functions, a 32-bit adder and a mux to select the function.

***Figure 4 -** ALU Structure*



The alu_function[6:0] bus in the core is a concatenation of the individual control signals in the ALU. The following table describes these control signals.

Table 1 ALU Function Encoding

Field	Function
swap_sel	Swaps the a and b inputs
not_sel	Selects the NOT version of b
cin_sel[1:0]	Selects the carry in to the full added from { c_in, !c_in, 1, 0 }. Note that bs_c_in is the carry_in from the barrel shifter.
cout_sel	Selects the carry out from { full_adder_cout, barrel_shifter_cout }
out_sel[2:0]	Selects the ALU output from { 0, b_zero_extend_8, b, and_out, or_out, xor_out, full_adder_out }

2.2 Pipeline Operation

2.2.1 Load Example

The load instruction causes the pipeline to stall for two cycles. This section explains why this is necessary. The following is a simple fragment of assembly code with a single load instructon with register instructions before and after it.

```
0  mov r0, #0x100
4  add r1, r0, #8
8  ldr r4, [r1]
c  add r4, r4, r0
```

The table below shows which instruction is active in each stage of the processor core for each clock tick. When the core comes out of reset the execute stage starts generating fetch addresses. It starts at 0 and increments by 4 each tick. In tick 1 the first instruction, at address 0, is fetched, This simple example assumes that all accesses are already present in the cache so fetches only take 1 cycle. Otherwise read accesses on the wishbone bus would add additional stalls and complicate this example.

At tick 2 the first instruction, 0, is decoded and at tick 3 it is executed. This means that the r0 register, which is the destination for instruction 0, does not output the new value until tick 4, where it is used as an input to the second instruction.

At tick 5 the load instruction, instruction 8, stalls the decode stage. In the execute stage it calculates the load address and this is used by the fetch stage in tick 6. Also in tick 5 the instruction c is saved to the pre_fetch_instruction register. This is used once the load instruction has finished and its use saves needing an additional stall cycle to reread instruction c.

At tick 6 the value at address 0x108 is fetched and at tick 7 it is written into r4. The new value of r4 is then available for instruction c in tick 8.

Table 2 Pipeline load example

Stage	Tick 0	Tick 1	Tick 2	Tick 3	Tick 4	Tick 5	Tick 6	Tick 7	Tick 8
Fetch									
address	-	0	4	8	c	10	108	10	14
access type		read	read	read	read	read, ignored	read	read	read
Decode									
instruction	-	-	0	4	8	8	8	c	10
pre_fetch_instruction	-	-	-	-	-	[c]	[c]		
Execute									
instruction	-	-	-	0	4	8	8	8	c
address_nxt	0	4	8	c	10	108	10	14	18

2.2.2 Store Example

The store instruction also causes the pipeline to stall for two cycles. This section explains why this is necessary. The following is a simple fragment of assembly code with a single store instructon with register instructions before and after it.

```
0  mov r0, #0x100
4  mov r1, #17
8  str r1, [r0]
c  add r1, r0, #20
```

The table below shows which instruction is active in each stage of the processor core for each clock tick. At tick 5 the store instruction, instruction 8, stalls the decode stage. In the execute stage it calculates the store address and this is used by the fetch stage in tick 6. Also in tick 5 the instruction c is saved to the pre_fetch_instruction register. This is used once the store instruction has finished and its use saves needing an additional stall cycle to reread instruction c. In tick 7 the instruction after the store instruction is decoded and in tick 8 it is executed.

Table 3 Pipeline store example

Stage	Tick 0	Tick 1	Tick 2	Tick 3	Tick 4	Tick 5	Tick 6	Tick 7	Tick 8
Fetch									
address	-	0	4	8	c	10	100	10	14
access type		read	read	read	read	read, ignored	write	read	read
Decode									
instruction	-	-	0	4	8	8	8	c	10
pre_fetch_instruction	-	-	-	-	-	[c]	[c]		
Execute									
instruction	-	-	-	0	4	8	8	8	c
address_nxt	0	4	8	c	10	100	10	14	18