MiniGPU

ISA

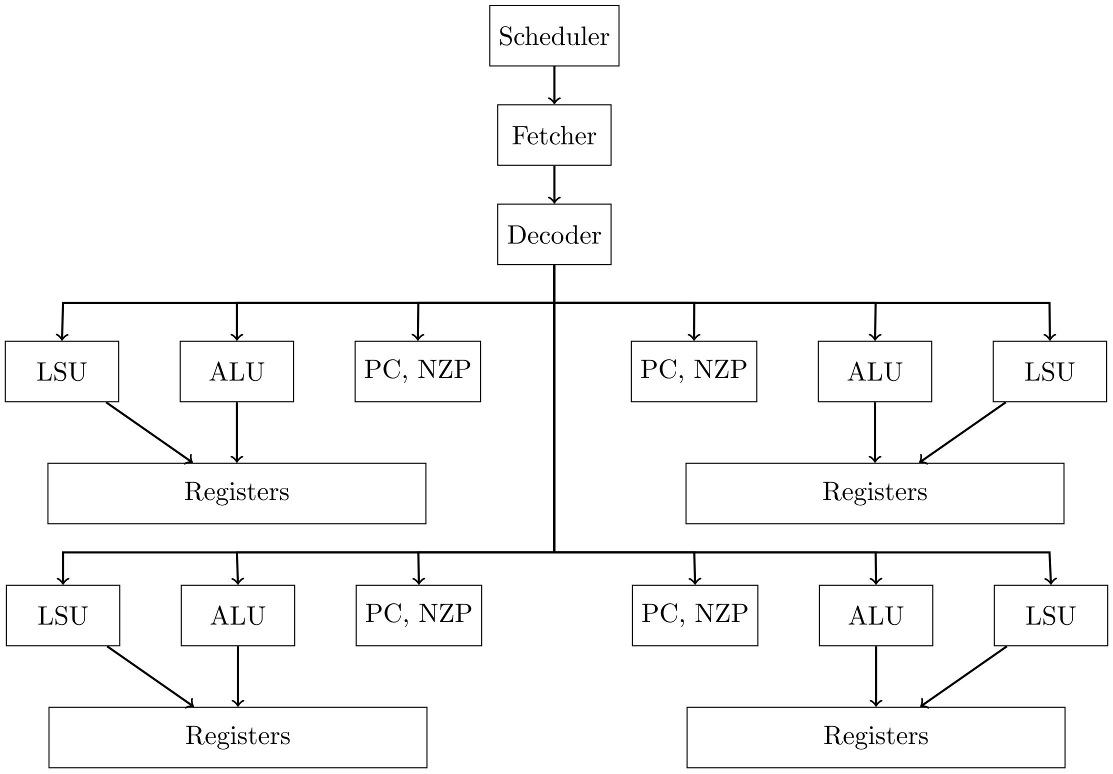
Table 1: Instruction Set Summary

Mnemonic	Semantics	Encoding
NOP	PC = PC + 1	0000 `xxxx xxxx xxxx`
BRnzp	NZP ? PC = `IMM8`	0001 `nzpx iiii iiii`
CMP	NZP = sign(`Rs - Rt`)	0010 `xxxx ssss tttt`
ADD	`Rd = Rs + Rt`	0011 `dddd ssss tttt`
SUB	`Rd = Rs - Rt`	0100 `dddd ssss tttt`
MUL	`Rd = Rs * Rt`	0101 `dddd ssss tttt`
DIV	`Rd = Rs / Rt`	0110 `dddd ssss tttt`
LDR	`Rd = data_mem[Rs]`	0111 `dddd ssss xxxx`
STR	`data_mem[Rs] = Rt`	1000 `xxxx ssss tttt`
CONST	`Rd = IMM8`	1001 `dddd iiii iiii`
RET	done	1111 `xxxx xxxx xxxx`

Specifications

The GPU consists of 2 cores, and each core has 4 threads. These 4 threads form a block.

Each core is expected to follow the rough structure shown below. Since every core must be capable of processing 4 threads simultaneously, we have 4 copies of the crucial components.



The device control register, dispatcher, memory controllers, and the memory itself sit outside the core, and required signals from those components to the core can be given based on the description listed below.

Datapath between components in a thread, i.e. between the PC, NZP register, LSU, ALU and the registers will also have to be made from the description given below. All wires and control signals are stated clearly.

Components required

0.0.1 ALU

Purpose

The ALU executes arithmetic operations and sets NZP flags based on comparisons. It is shared across all threads but each thread has logical access to its own ALU behavior.

Inputs and Outputs (Logical Description)

`Inputs:`

```
clk: Clock signal

reset: Synchronous reset

enable: Triggers ALU computation

core_state: Signals EXECUTE phase (binary 101)

decoded_alu_arithmetic_mux: [1:0] opcode: 00=ADD, 01=SUB, 10=MUL, 11=DIV

decoded_alu_output_mux: 0=Arithmetic result, 1=NZP flags

rs, rt: Operands (8-bit unsigned)

Output:

alu_out: 8-bit result or {00000, N, Z, P}
```

Computation Logic

- If `reset` is active: clear the output to 0
- If `enable` is active and `core-state == 3'b101`:
  - If `decoded-alu-output-mux == 1`, then:
    - Compute NZP flags:
      - N = (rs - rt < 0)
      - Z = (rs == rt)
      - P = (rs - rt > 0)
    - Output = 00000, N, Z, P
  - Else, compute arithmetic based on mux selector:
    - ADD: rs + rt
    - SUB: rs - rt
    - MUL: rs * rt (lower 8 bits)
    - DIV: rs / rt (undefined behavior if rt = 0)

Internal Notes

- This module uses an internal 8-bit register to hold the output
- All logic is edge-triggered on the positive edge of `clk`
- Flags are encoded only when `output-mux` is set to 1

0.0.2 Register File

The GPU ises a register file with 16 registers, each register being 8 bits wide. Each register is addressed with 4 bits.

It is advised to store constants crucial to the GPU working in the register file too, so that it can be easily accessed by other components. Towards this, we suggest dedicating 3 registers, one for `block_id`, one for `thread_id` and one for the constant `threads_per_block`.

The regsiter file will have to take in the `block_id`, `thread_id` and `threads_per_block` as inputs. There are two output channels, and one input channel. Hence, there are three address inputs, one data input, and two data outputs. It goes without saying that inputs like clock, reset and enable are also required. The register file also takes the state of the scheduler as an input, and only functions if the core is in the `REQUEST` state.

Writeback into the register file can come from the ALU, LSU, or even the immediate. A MUX to select which of the data has to be written can be incorporated.

0.0.3 Program Counter & NZP Register

We give the PC module two functions: to update the program counter and to store the appropriate conditons in the NZP Register (see BrNZP). Only when the scheduler state is `EXECUTE`, will the PC get updated. We have to input the current PC and the module outputs the updater PC. In the module, check the NZP flags for choosing between linear or branched flow. Control signals that determine what the NZP flag must be should be passed as an input. The address to branch to comes from the immedaite, which also has to be an input to the PC.

During the `UPDATE` state, the NZP register has to be updated with the negative, zero, positive flags (a 3-bit register) based on the ALU output. As a consequence, the output of the ALU must also be an input to the PC. Consequently, an enable pin for the register must also be given as input.

0.0.4 Decoder

Decodes an instruction into all the control signals for various components. The decoder is synchronous, i.e. the modifications to the control signals if a new instruction is passed takes place only at a clock edge.
It takes the following as an input

- **Clock**
- **Reset**
- **Core State**: 3 bit wide signal for the state of the scheduler
- **Instruction**: 16 bit wide signal that carries the relevant instruction

For ease of reference, the following memory signals need to be decoded:

- **RD Address**
- **RS Address**
- **RT Address**
- **Decoded NZP**
- **Immediate**

The following control signals for the components are to be decoded:

- **Register Write Enable**
- **Memory Read Enable**
- **Memory Write Enable**
- **NZP Write Enable**
- **Register Input MUX**
- **ALU Control**
- **ALU Output MUX**
- **Next PC MUX**

0.0.5 Memory Controller

Purpose

This module schedules and executes memory requests (reads and writes) coming from multiple consumers. It supports a configurable number of concurrent memory channels.

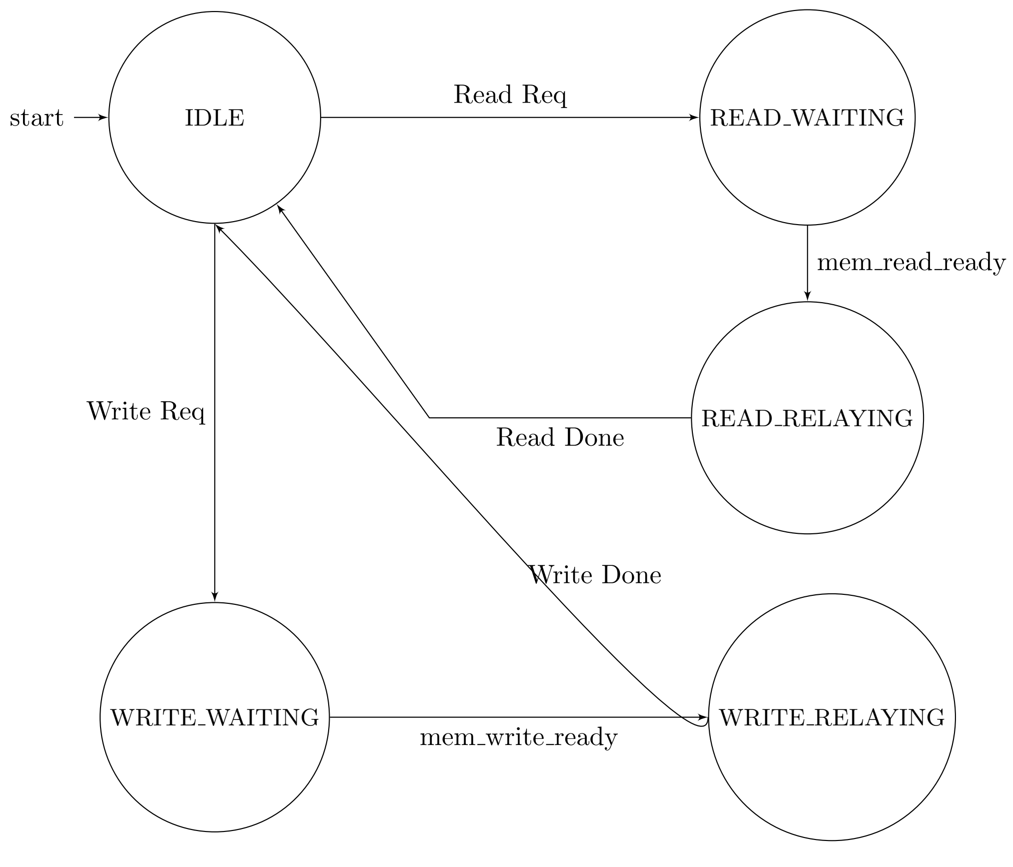
Parameters and Interface Summary

```
Parameters:

ADDR_BITS: Address width

DATA_BITS: Data width

NUM_CONSUMERS: Number of consumer cores or LSUs

NUM_CHANNELS: Number of concurrent memory channels

WRITE_ENABLE: If 1, writing to memory is allowed

Interface:
```

```
Receives: consumer_read_valid, consumer_write_valid + corresponding addresses/data

Sends: data and handshakes to memory (read/write)

Returns: read data to correct consumer, signals ready
```

Channel FSM Design

Each channel operates independently with its own FSM.



Logic Description

- For each channel:
  - Look through all consumers to find a valid read/write request
  - If found:
    - Issue memory command (read or write)
    - Mark consumer as *being served* to avoid duplication

  - Wait for memory to return valid ready signal
  - Relay data (or acknowledge write) back to the consumer
  - Once consumer de-asserts valid signal, return to IDLE
- Prevents multiple channels from serving the same consumer at once using a global bitmap

Behavioral Summary

- Implements a greedy scheduler per channel
- Ensures memory bandwidth is respected using `mem_read_ready` / `mem_write_ready`
- Priority order of consumers is static and always starts from consumer 0

0.0.6 LSU

Purpose

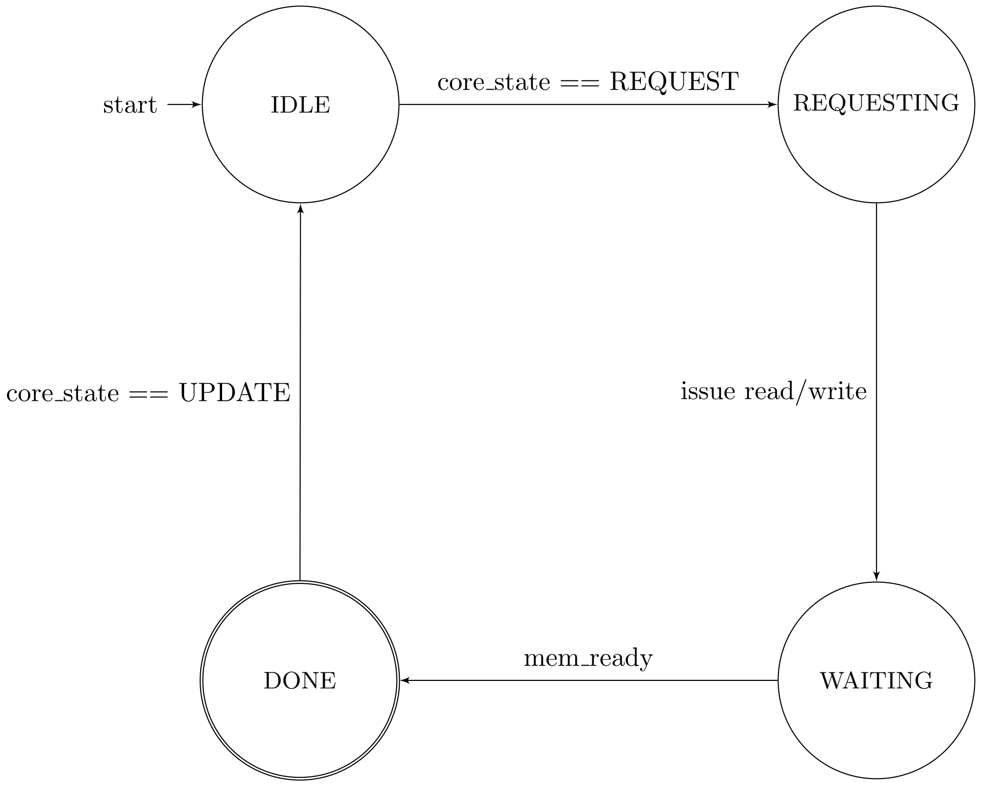
The LSU allows each thread to asynchronously interact with memory. It issues and waits for either load (LDR) or store (STR) instructions during the appropriate core states.

Logical Interface

```
Inputs:

clk, reset: Clock/reset signals

enable: Triggers LSU operation for a thread

core_state: Controls progression through the memory access pipeline

decoded_mem_read_enable: 1 if LDR

decoded_mem_write_enable: 1 if STR

rs: Address register

rt: Data register (used for store only)

Memory Interface:

mem_read_valid, mem_read_address, mem_read_ready, mem_read_data

mem_write_valid, mem_write_address, mem_write_data, mem_write_ready

Output:

lsu_out: Result of memory read
```

`lsu_state: Current LSU FSM state (2 bits)`

FSM for LDR/STR Instructions



Behavior Description

- LSU begins when `enable = 1` and `decoded_mem_read/write_enable = 1`
- In IDLE:
  - Wait for core state to be REQUEST (3'b011)
- In REQUESTING:
  - Assert memory valid signals and supply addresses (and data if store)
- In WAITING:
  - Await memory ready
  - Capture read result in `lsu_out` (if LDR)
- In DONE:
  - Wait for core state to be UPDATE (3'b110), then return to IDLE

0.0.7 Device Control Register

Purpose

The DCR provides a simple software-configurable mechanism to control device parameters. In this example, it sets the number of threads to be launched in a kernel.

Interface Summary

```
Inputs:

clk: Clock signal

reset: Synchronous reset

device_control_write_enable: Triggers register update

device_control_data: 8-bit data to be stored

Output:

thread_count: 8-bit output reflecting current configuration
```

Behavior Description

- On reset: the thread count is cleared to zero.
- On rising edge of `clk`:
  - If `device_control_write_enable` is high:
    - Capture `device_control_data` and store in internal register
    - Output value is immediately reflected on `thread_count`
- If write is not enabled, register retains its previous value.

Usage Notes

- This register is typically written by software just before kernel execution.
- Can be extended to configure other global parameters like block size or kernel ID.
- Simple synchronous latch with overwrite logic.

0.0.8 Dispatcher

Purpose

The dispatcher coordinates the distribution of work (blocks of threads) across available compute cores. It ensures that the entire thread pool is processed and indicates when kernel execution is fully completed.

Interface Summary

`Parameters:`

`NUM_CORES: Number of compute cores`

`THREADS_PER_BLOCK: Number of threads assigned to each block`

`Inputs:`

`clk, reset: Clock and synchronous reset`

`start: Trigger signal to initiate kernel execution`

`thread_count: Total number of threads to execute`

`core_done: Per-core signal indicating a block has finished`

`Outputs:`

`core_start: Launch signal for each core`

`core_reset: Reset signal for each core before new block dispatch`

`core_block_id: Block ID assigned to each core`

`core_thread_count: Threads assigned to a block (used for final block truncation)`

`done: Global signal indicating all blocks have been processed`

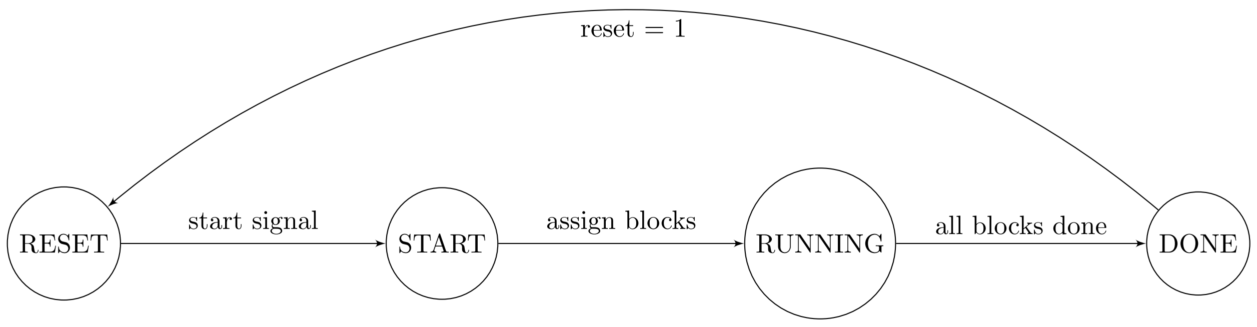
Block Distribution Logic

- Calculate total number of blocks as:

  `total_blocks = (thread_count + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK`

- Maintain counters:
  - `blocks_dispatched` — how many blocks have been issued
  - `blocks_done` — how many blocks have been completed

Dispatch FSM Behavior



Core Control Logic

- At **reset**: all counters and core signals are cleared, and cores are put in reset.
- At rising edge of `start`:
  - All cores are released from reset
  - Begin dispatching blocks to idle cores
- When a core completes a block (`core_done` high):
  - Increment `blocks_done`
  - Reset the core and dispatch a new block if available
  - If it is the last block, set `done = 1`
- The final block may have fewer threads; use modulo logic to adjust `core_thread_count`

Edge Cases and Notes

- All cores operate independently; dispatcher schedules based on core availability
- Assumes compute cores automatically clear `core_done` after reset

0.0.9 Scheduler

Each core has it's own scheduler where multiple threads can be processed with the same control flow. It is modelled as an FSM, which coordinates instruction fetching, decoding, memory operations, execution, and state updates across multiple threads in a SIMD fashion.

Remarks:
SIMD Execution: All threads in block execute same instruction stream
Memory Synchronization: WAIT state ensures memory ops complete before execution
Branch Handling: Current implementation assumes no divergence (all threads take same path)
Thread Management: Processes 4 threads simultaneously (4 threads per block)

Control Signals: Coordinates with external fetch and LSU modules through inputs to scheduler

The inputs to the FSM include:

- **Clock**
- **Reset**
- **Start**: Starts the processing of a block when set
- **Memory Read Enable**: Decoded signal from decoder when read required.
- **Memory Write Enable**: Decoded signal from decoder when write required.
- **Decoded Return**: Indicates if the instruction is a return.
- **Fetcher State**: 3 bit wide signal that indicates the current state of the fetcher.
- **LSU State**: 2 bit wide signal from each thread that indicates the state of the LSU.
- **Next PC**: 8 bit wide signal per thread. While we will not need it in the naive implementation, it is needed in case threads from the same block diverge to different instructions.

The outputs of this module are:

- **Current PC**
- **Scheduler State**
- **Done**: An output signal when the scheduler is done.

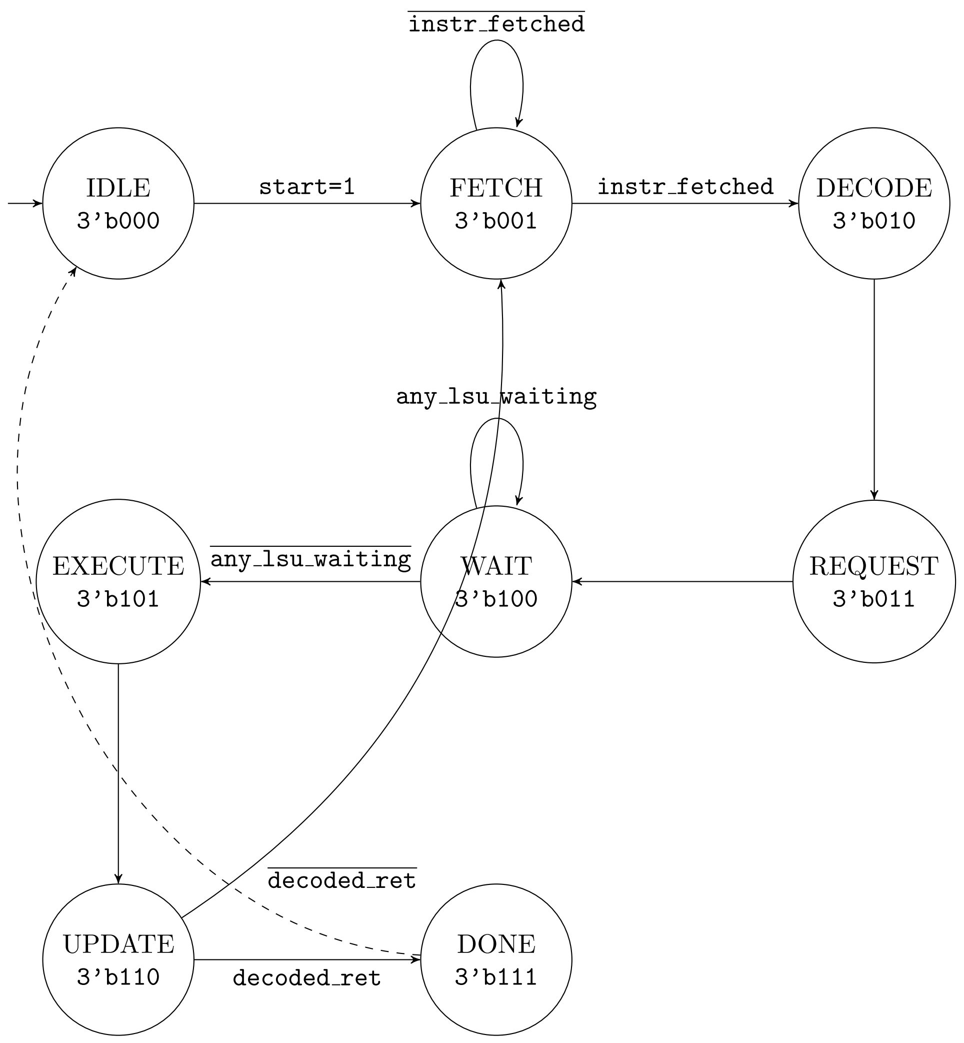


Figure 1: GPU Scheduler State Transition Diagram

The detailed operation of each of the states is then given below:

State	**Cycles**	**Operations**
`IDLE`	> 1	• Reset all internal registers • Wait for `start` signal • Initialize `current_pc` to entry point
`FETCH`	> 1	• Send `current_pc` to instruction memory • Wait until `fetcher_state == FETCHED` • Store fetched instruction in pipeline register
`DECODE`	1	• Decode instruction fields • Set control signals: `decoded_mem_read_enable = opcode == LOAD;` `decoded_mem_write_enable = opcode == STORE;` `decoded_ret = opcode == RETURN;`
`REQUEST`	1	• Activate LSUs if memory operation needed • For stores: Send address/data to memory controller • For loads: Send address to memory controller
`WAIT`	> 1	• Poll all LSU states: `any_lsu_waiting = (lsu_state == REQUESTING ||` `lsu_state == WAITING);` • Stall until all memory ops complete
`EXECUTE`	1	• Perform ALU operations • Calculate branch targets • Compute `next_pc` for each thread
`UPDATE`	1	• Update register file • Set condition codes (NZP) • Commit `next_pc` values: `if (!decoded_ret)` `current_pc <= converge(next_pc);`

0.0.10 Fetcher

This module implements an instruction fetcher for a GPU core that retrieves instructions from program memory based on the current program counter (PC) value.

It is implemented as an FSM with three states, `IDLE`, `FETCHING` and `FETCHED`

The following are inputs to the fetcher:

- **Current PC**: PC of the instruction that shall be fetched
- **Scheduler state**: Output from the scheduler
- **Memory Read Ready**: Signal from memory to initiate read
- **Memory Read Data**: Data that is read from the program memory

The following are outputs of the fetcher:

- **Memory Read Address**: Basically current PC
- **Fetcher state**: 3 bit signal indicating the state of the fetcher
- **Memory Read Valid**: Handshake signal between fetcher and memory. Indicates that a valid read active.
- **Instruction**: 16 bit wide instruction fetched from program memory

State	**Cycles**	**Operations**
`IDLE`	≥ 1	• Default state after reset • All outputs held at zero (`mem_read_valid=0`, `instruction=0x0000`) • Waits for `core_state` to enter `FETCH` (3'b001) • Immediately transitions to `FETCHING` when triggered
`FETCHING`	≥ 1	• Asserts `mem_read_valid` and outputs `current_pc` on `mem_read_address` • Holds request until memory acknowledges with `mem_read_ready` • On acknowledgment: – Latches `mem_read_data` into `instruction` – Deasserts `mem_read_valid` – Transitions to `FETCHED` • Stalls indefinitely if memory doesn't respond
`FETCHED`	≥ 1	• Holds fetched `instruction` stable for core to consume • Waits for `core_state` to enter `DECODE` (3'b010) • On decode phase start: – Transitions back to `IDLE` – Maintains `instruction` output until next fetch