APARA Engine: Microarchitecture and Timing

Madhav Desai
Department of Electrical Engineering, IIT Bombay
Scribe: Neha Karanjkar, IIT Goa

June 10, 2026

Contents

1 Introduction and Architectural Overview **1**
- 1.1 Scope and companion documents . . . 1
- 1.2 System context . . . 2
- 1.3 Engine organisation at a glance . . . 2
- 1.4 Threads and the execution model . . . 3
- 1.5 Thread and register model . . . 4
- 1.6 Instruction bundles . . . 5
- 1.7 Memory model . . . 6
- 1.8 Document organisation and conventions . . . 6

2 Components and Interfaces **9**
- 2.1 External interfaces of the engine . . . 10
- 2.2 TEU pipeline stages . . . 10
- 2.3 Inter stage channels . . . 10
- 2.4 CCU register map . . . 11

3 Microarchitecture **14**
- 3.1 Per thread state . . . 14
- 3.2 Pipeline overview . . . 15
- 3.3 FETCH stage . . . 16
  - 3.3.1 NOP padding . . . 17
  - 3.3.2 Branch predictor . . . 17
- 3.4 DECODE stage . . . 18
- 3.5 REGFILE stage . . . 19
  - 3.5.1 Register file locks . . . 19
  - 3.5.2 Picker . . . 19
  - 3.5.3 Side ports . . . 20
- 3.6 EXEC stage . . . 20

3.6.1 Instruction slot alignment constraints 20
3.6.2 Memory operations 21
3.6.3 CTI handling and stream id 21
3.6.4 Exception detection 22
3.6.5 RAS state and operations 23
3.6.6 Vector parallelism 23
3.7 WRITEBACK stage 23
3.7.1 Mispredict signalling 24
3.7.2 HALT 24
3.7.3 Lock release 24
3.8 Memory subsystem 25
3.8.1 Data memory (DMEM) 25
3.8.2 Instruction memory (IMEM) 26

4 Timing **27**
4.1 Cycle model 27
4.2 Per stage cycle behaviour 27
4.3 Per opcode latency 27
4.4 Stall and squash behaviour 28

5 Opcode Numbering Discrepancy **31**

6 Open Items **33**

Abstract

This document describes the microarchitecture and timing model of the APARA edge compute engine: its block-level organisation, the components of the execution pipeline, the per-thread state, the cycle-level behaviour of each pipeline stage, and the per-opcode timing model. It is intended to be reasonably self-contained for the purpose of building a cycle-approximate simulation model of the engine.

The instruction-set and programmer's view (instruction encodings, per-instruction semantics, assembly syntax, sample programs) are not reproduced here; they are covered in the companion document *AparaReference* and are only summarised where needed to follow the microarchitecture. The sources used are the `hsys` structural files, the `McodeMachine` reference C model, the `Aa_model` hardware description, and the designer's notes.

Items that are not yet fully pinned down are flagged [**TO CONFIRM:** like this] throughout the text and are collected together in the final chapter (Chapter 6). Answers found in the `Aa_model` hardware description source are shown in [**Answer:** blue] inline next to the original query, and new findings from the `Aa_model` are marked in teal.

Chapter 1

Introduction and Architectural Overview

The APARA engine is a four-threaded VLIW compute engine. Each thread executes a stream of instruction bundles. A bundle contains 1, 2, 4, or 8 instructions of 32 bits each. The instructions in a bundle are intended to be executed in parallel. Successive bundles of one thread are executed in program order. The four threads are time multiplexed onto a single execution pipeline.

The engine is built around a five-stage pipeline. The five stages are called FETCH, DECODE, REGFILE, EXEC, and WRITEBACK in this document. The instruction memory and data memory are local to the engine. A control unit serves as the bridge between the host and the engine, and a DMA engine moves data between the engine memories and the system memory.

1.1 Scope and companion documents

This document is the microarchitecture and timing reference for the engine. It describes *how* the engine is built and *how long* operations take. It deliberately does not reproduce the instruction-set architecture in full. The following companion documents cover the programmer's view and should be consulted for those details:

- *AparaReference* (`docs/AparaReference.pdf`): the full instruction-set reference — per-instruction syntax, operand encodings, data types, and

worked assembly examples. This is the authoritative source for instruction semantics.

- *Accelerator* (`docs/Accelerator.pdf`): the designer's slide deck giving the high-level concept and a tour of the instruction classes.
- The `McodeMachine` reference C model: the functional golden model (instruction-accurate, not cycle-accurate).

Section 1.5 through Section 1.7 below summarise just enough of the architecture — the thread and register model, the bundle format, the instruction classes, and the memory map — to make the rest of this document readable on its own. For anything beyond that summary, refer to *AparaReference*.

1.2 System context

The APARA accelerator is a memory-coupled accelerator that attaches to a system interconnect alongside a management (host) processor and system memory, as shown in Figure 1.1. The host processor is the master: it configures the engine, loads its instruction memory, starts and observes threads, and programs DMA transfers, all over the configuration bus (the *AFB*, Accelerator Front-side Bus). The engine exposes a master port (M) for DMA traffic to system memory and a slave port (S) for host access into the engine's memories. In a full SoC the accelerator is replicated many times across clusters; the wider chip context (a multi-cluster, many-core device) is described in *AparaReference* and is out of scope here.

1.3 Engine organisation at a glance

Figure 1.2 shows the block-level organisation of a single engine. The shared execution pipeline reads instructions from a local 16 KB instruction memory and operands from a local 64 KB, two-bank data memory. Four independent thread-state blocks (each holding a program counter, register set, return-address stack, and error register) are multiplexed onto this one pipeline by the thread control unit. The control interface lets a configuration processor program the instruction memory, set up and monitor threads, and program the DMA engine, which moves data between the engine's data memory and

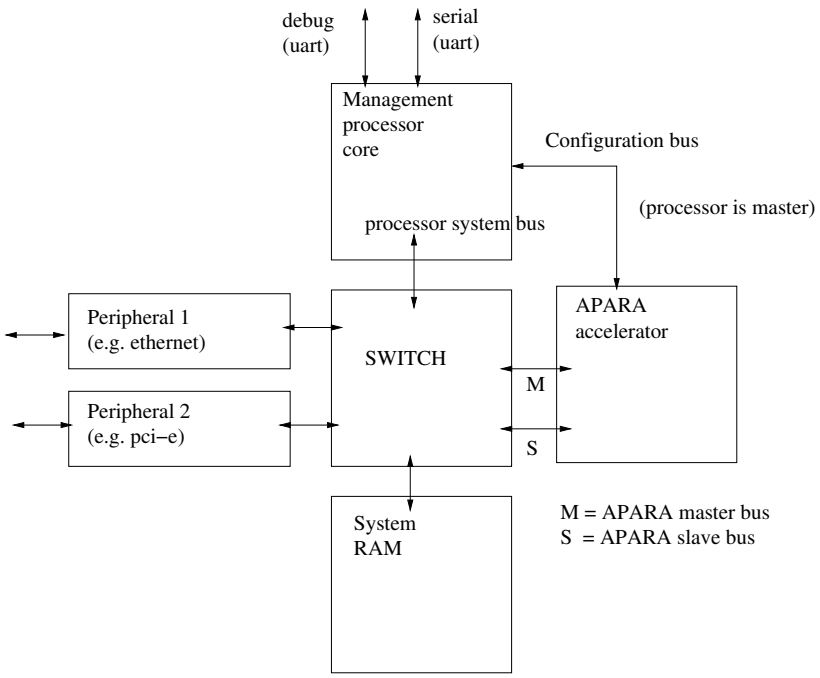


Figure 1.1: System context: the APARA accelerator on a system switch with a management processor and system memory. M is the APARA master bus (DMA to system memory), S is the APARA slave bus (host access into engine memory).

system memory across the switch. The components are enumerated in Chapter 2 and their internal behaviour in Chapter 3.

1.4 Threads and the execution model

Up to four threads are active at once. Each thread has its own private state — program counter, register set, return-address stack, and error register — and executes its own bundle stream from the shared instruction memory. The threads are interleaved onto the single execution pipeline: on each opportunity FETCH picks one active, non-halted thread (round robin) and injects its next bundle. The abstract execution algorithm is shown in Figure 1.3; the hardware realises this as a pipelined, multi-threaded version of the same loop.

Because the four threads are independent, the pipeline can stay busy even when one thread stalls (for example on a long-latency memory access or a branch mispredict): the other threads continue to make progress. Within a single thread, bundles are issued strictly in program order. The detailed per-stage behaviour is given in Chapter 3.

1.5 Thread and register model

The architecturally visible state of each thread is:

- **PC**: a 32-bit program counter (a word index into the instruction memory).
- **Register set**: 32 general-purpose registers $R_0 \dots R_{31}$, each 64 bits wide. R_0 always reads as zero and writes to it are dropped.
- **RAS**: a return-address stack used by CALL/RETURN.
- **ERROR**: a 32-bit error/status register.

The hardware layout of this state (RAS depth, ERROR bit-fields, host visibility) is detailed in Section 3.1 and Section 2.4.

A register's 64 bits can be interpreted under a range of data types, selected per instruction by a type field. The supported types are summarised in Table 1.1. The 128- and 256-bit types are used only to describe memory transfers spanning two or four registers; they are not arithmetic types.

Table 1.1: Data types (see *AparaReference* for full details).

Group	Types
Signed integer	`$i4, $i8, $i16, $i32, $i64`
Unsigned integer	`$u4, $u8, $u16, $u32, $u64`
Wide (memory only)	`$u128` (2 regs), `$u256` (4 regs)
Float	`$f4, $f8, $f16, $f32, $f64`
Vector integer	`$vi4, $vi8, $vi16, $vi32` (and unsigned `$vu*`)
Vector float	`$vf4, $vf8, $vf16, $vf32`

1.6 Instruction bundles

Each instruction is 32 bits wide. The most-significant bit (bit 31) is the *bundle-start* bit: it is set on the first instruction of a bundle. The opcode always occupies bits 30:26. The remaining bits encode the destination register, source registers or immediate, and the type field, depending on the instruction class (Figure 1.4; see *AparaReference* for the exact field encodings of each instruction).

The bundle-start bits delimit bundles within the instruction stream. A bundle holds 1, 2, 4, or 8 instructions, and its start address is aligned to its size (an 8-instruction bundle starts at a multiple of 8 words, a 4-instruction bundle at a multiple of 4, and so on). At fetch time a 256-bit (8-instruction) memory line may therefore hold a single 8-instruction bundle, or several smaller bundles (Figure 1.5); FETCH sequentialises them in PC order (Section 3.3).

A bundle has limits on which instruction classes may appear and where: at most one control-transfer instruction per bundle, no two instructions writing the same register, and slot-placement rules for control-transfer, divide/square-root, and memory instructions (Section 3.6.1). The instruction classes are:

- **Scalar arithmetic/logic**: integer $+, -, \times, /$, shifts, and bitwise and/or/xor/nand/nor/xnor; floating-point $+, -, \times, /$ and square-root.
- **Vector arithmetic**: vector add/subtract/multiply/scale, dot-product, and vector-reduce.
- **Data transfer**: load/store, 1 to 32 bytes per transfer, with optional auto-increment.
- **Control transfer**: branch, call, return, and conditional move (CMOV).
- **Casts**: scalar and vector type conversions.
- **Miscellaneous**: set, slice, pack, null, nop, halt.

See *AparaReference* for the operands, immediate formats, and exact semantics of each instruction.

1.7 Memory model

The engine has two disjoint local memories:

- **Instruction memory (IMEM)**: 16 KB, holding up to 4096 32-bit instructions. The execution pipeline fetches 256-bit (8-instruction) aligned lines.
- **Data memory (DMEM)**: 64 KB, organised as two 32 KB banks, each with a 256-bit-wide port. With both banks active the pipeline can move up to 64 bytes per cycle to/from memory.

Memory addresses computed by the pipeline are 32 bits wide internally. A wider 36-bit system address is used at the engine boundary (see the terminology note in Section 1.8); accesses that fall outside the engine's local 64 KB are routed off-island through the DMA engine. The microarchitecture of the memory subsystem — bank selection, port assignment, and off-island routing — is detailed in Section 3.8.

1.8 Document organisation and conventions

The rest of this document is organised as follows. Chapter 2 describes the components of the engine and the interfaces between them, with explicit references to the `hsys` files where each item is defined. Chapter 3 describes the per-thread state and the operation of the five pipeline stages. Chapter 4 gives the timing model used by the simulation, including per-opcode latency and the behaviour of stalls and squashes. Chapter 5 records an opcode-numbering discrepancy that must be resolved before encoding the opcode-to-latency map. Chapter 6 collects all open items.

A note on terminology and conventions. The engine-internal address space is 32 bits wide. The system-wide address space is 36 bits, with the upper 4 bits holding an address-space identifier (ASI); engine-internal nets carry 32-bit addresses. Throughout the document, open questions are flagged [**TO CONFIRM:** like this], answers subsequently found in the `Aa_model` hardware description are shown [**Answer:** like this], and additional findings from the `Aa_model` are marked [**New finding from Aa_model:** like this]. All such open items are gathered in Chapter 6.

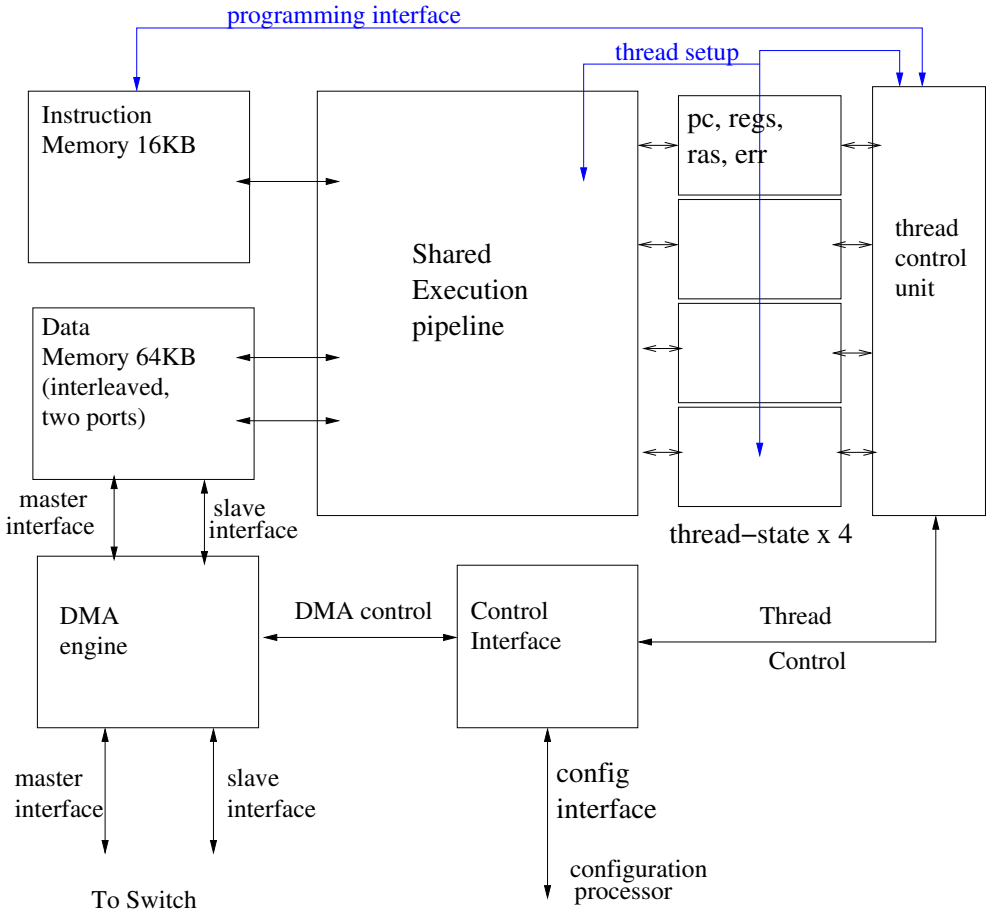


Figure 1.2: Block-level organisation of one APARA engine: a shared execution pipeline, four thread-state blocks, local instruction and data memories, the DMA engine, and the control interface to the configuration processor.

```
// 4 threads: 4x PC, 4x REGSET, 4x RAS, 4x ERR
PC[1..4]  = init values;   REGS[1..4] = -;
RAS[1..4] = empty;         ERR[1..4]  = 0;   ACTIVE = 1;

while (1)
{
    bundle := fetch_bundle_from_imem(PC[ACTIVE]);
    ops    := get_ops_needed_by_bundle(bundle, REGS[ACTIVE]);
    out    := execute_instructions_in_bundle(bundle, ops);
    PC[ACTIVE], REGS[ACTIVE], RAS[ACTIVE], ERR[ACTIVE]
           := update_pc_regs_ras_error(out);
    ACTIVE := next_active(ACTIVE, ERR[1..4]);
    if (ACTIVE == 0) break;
}
```

Figure 1.3: Abstract multi-thread execution algorithm (after *AparaReference*). The hardware pipelines this loop and interleaves the four threads onto one execution pipeline.

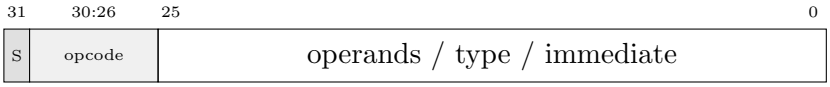


Figure 1.4: 32-bit instruction word. Bit 31 (S) is the bundle-start bit; bits 30:26 are the opcode; the remaining bits encode operands, type, and immediates according to the instruction class.

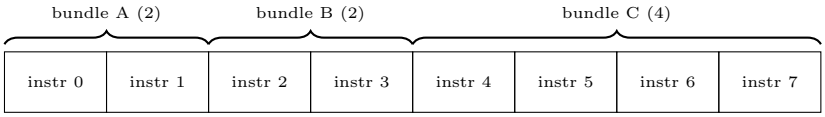


Figure 1.5: A 256-bit instruction line holds eight 32-bit instruction slots. Bundle-start bits partition the line into bundles; here three bundles of sizes {2, 2, 4}. FETCH emits them one per cycle in PC order.

Chapter 2

Components and Interfaces

The engine is described structurally in the file `engine.hsys`. It contains six children. The function of each is summarised in Table 2.1; their block-level arrangement was shown in Figure 1.2 (Chapter 1). The common short names introduced in the rightmost column are used in the rest of this document.

Table 2.1: Engine children and the names used in this document.

hsys instance	hsys file	common name	role
`ccu_mt_inst`	`ccu_mt.hsys`	CONTROL_UNIT	host facing per thread control and observation
`teu_inst`	`teu.hsys`	TEU	the five stage pipeline that executes bundles
`dma_inst`	`dma.hsys`	DMA	moves data between engine memories and system memory
`dmem_inst`	`data_mem.hsys`	DMEM	64 KB local data memory
`imem_inst`	`instr_mem.hsys`	IMEM	16 KB local instruction memory
`afb_demux_inst`	`afb_engine_demux.hsys`	AFB demux	routes AFB requests to CONTROL_UNIT, DMA, or imem init

The full forms of the abbreviations introduced above are:

- CCU: Configuration and Control Unit. There are four per engine, one per thread, packaged together as `ccu_mt`.
- TEU: Thread Execution Unit.

- AFB: Accelerator Front side Bus, the bus the host uses to talk to the engine.
- ACB: Accelerator Cache Bypass bus, used for cache bypassed data paths.
- ASI: Address Space Identifier (4 bits).
- RAS: Return Address Stack.
- PC: Program Counter.
- NPC: Next Program Counter.
- CTI: Control Transfer Instruction (branch, call, return).

2.1 External interfaces of the engine

The engine block in `engine.hsys` declares the input and output ports listed in Table 2.2. These are the only nets that cross the engine boundary.

2.2 TEU pipeline stages

The TEU is described in `teu.hsys`. It contains five stage instances, one per pipeline stage. Table 2.3 lists them along with the common name used in the rest of this document.

2.3 Inter stage channels

All channels between TEU pipeline stages are elastic. They are FIFO buffered with the depths declared in the hsys files. Most are `$noblock $pipe` which means a writer stalls until the FIFO has space and a reader returns immediately with a valid bit clear if the FIFO is empty.

Table 2.4 lists the channels. Channel widths and depths are taken verbatim from the hsys declarations.
The TEU also has two memory ports to DMEM. They are declared in `teu.hsys` as `noblock_ajit_256_wide_bus_request_[0,1]` (307 bits, depth 2) for outgoing requests and `ajit_256_wide_bus_response_[0,1]` (257 bits,

Table 2.2: External pipes and signals of the engine (`engine.hsys`).

direction	name	kind	purpose
in	`CLUSTER_ID`	signal	cluster identifier
in	`ENGINE_ID`	signal	engine identifier within cluster
in	`INIT_BUNDLE_WORD_INDEX`	signal	start word index for thread 0
in	`ENABLE_ENGINE_RETIRE_TRACE`	signal	enables the retire trace pipe
in	`afb_request`	pipe	host configuration and DMA commands
in	`acb_dma_master_response`	pipe	system memory response to DMA reads
in	`acb_dma_slave_request`	pipe	host writes into engine memories
in	`noblock_afb_init_imem_request`	noblock pipe	direct IMEM init path (test)
out	`INTERRUPT_FROM_ENGINE`	signal	engine interrupt to host
out	`INTERRUPT_FROM_DMA`	signal	DMA interrupt to host
out	`afb_response`	pipe	replies to host configuration
out	`acb_dma_master_request`	pipe	DMA reads/writes to system memory
out	`acb_dma_slave_response`	pipe	host read replies
out	`afb_init_imem_response`	pipe	ack for direct IMEM init
out	`engine_retire_trace`	pipe	per cycle retire trace

depth 2) for incoming replies. Each port is 256 bits wide on the data side. The two ports are wired one per DMEM bank, see §3.8.

2.4 CCU register map

[**New finding from Aa_model:** The Aa_model source `cpu/ccu/docs/CCU.TXT` gives the full CCU register map. There are 8 registers per thread, listed in Table 2.5. The host accesses them via AFB using the CCU read/write commands. Register [5] is the exception status register and corresponds to the 32-bit ERROR register in the per-thread architectural state.]
Commands interpreted by the CCU: read imem (opcode 1), write imem (opcode 2), reset thread (opcode 3), run thread from PC (opcode 4), read thread register (opcode 5), write thread register (opcode 6), wait-on-complete / read

Table 2.3: TEU pipeline stages (`teu.hsys`).

hsys instance	hsys file	common name
`accel_fetch_inst`	`fetch.hsys`	FETCH
`accel_decode_inst`	`decode.hsys`	DECODE
`accel_regfile_v2_inst`	`register_file_v2.hsys`	REGFILE
`accel_exec_inst`	`exec.hsys`	EXEC
`accel_retire_inst`	`retire.hsys`	WRITEBACK

status (opcode 7).

Table 2.4: TEU inter stage channels (widths in bits, depths in entries).

from / to	name (hsys)	width	depth	purpose
IMEM → FETCH	`engine_imem_response`	257	2	256 bit instruction line plus valid
FETCH → IMEM	`noblock_engine_imem_request`	30	2	line aligned word index
FETCH → DECODE	`teu_fetch_to_decode`	336	2	one sequentialised bundle, including thread id, PC, stream id, instruction slots
DECODE → REGFILE	`noblock_teu_decode_to_register_file_read_request_0..3`	400	2	per thread bundle with decoded operand selectors
REGFILE → EXEC	`noblock_teu_register_file_to_exec`	2384	2	bundle with operand values for up to 8 instructions
EXEC → WRITE-BACK	`noblock_teu_exec_to_write_back_aggregated`	2842	1	aggregated results for the bundle
WRITEBACK → FETCH	`noblock_teu_pc_correction_pipe`	70	1	PC correction on mispredicted branch
WRITEBACK → FETCH	`noblock_teu_halt_pipe`	94	1	halt notification
WRITEBACK → REGFILE	`noblock_teu_regfile_to_retire`	123	2	writeback to clear locks and update file

Table 2.5: CCU internal registers per thread (`CCU.TXT`).

Index	Name	Contents
[0]	Control/status	bits [31:16] sequence number; [15:12] reserved; [11:8] command opcode; [7:0] flags
[1]	PC register	current program counter (word index)
[2]	Argument 0	imem address or register index
[3]	Argument 1	imem write data
[4]	Argument 2	imem write data
[5]	Exception status	ERROR register: bits [31:27] unused; [26] ras-error; [25] halt-flag; [24] exception-flag; [23:0] exception-codes (8 slots × 3 bits each)
[6]	Register high	high 32 bits of a 64-bit thread register read/write
[7]	Register low	low 32 bits of a 64-bit thread register read/write

Chapter 3

Microarchitecture

3.1 Per thread state

Each of the four threads has its own private state, introduced at the architectural level in Section 1.5 and illustrated in Figure 3.1. The state consists of:

- PC, the program counter, 32 bits.
- REGSET, 32 general purpose registers R_0 to R_{31}, each 64 bits wide. R_0 always reads as zero. Writes to R_0 are silently dropped.
- RAS, the return address stack. 64 entries deep, each 32 bits wide. A push on a full stack causes the thread to halt with the ERROR register set. There is no other overflow recovery mechanism.
- ERROR, a 32 bit error status register. [**TO CONFIRM:** The exact bit layout is not specified yet. As a working assumption, the least significant bits of ERROR mirror the error flag used by McodeMachine. The remaining bits are reserved.] [**Answer:** Confirmed from `CCU.TXT` and `fetch_core.aa`. Layout (Table 2.5, register [5]): bits [31:27] = unused; bit [26] = ras-error; bit [25] = halt-flag; bit [24] = exception-flag; bits [23:0] = exception-codes (8 instruction slots × 3 bits each, see §3.6.4). Visible to the host via CCU register [5].]

A thread can be in one of three observable states:

- Inactive: the thread does not exist or has not been started.

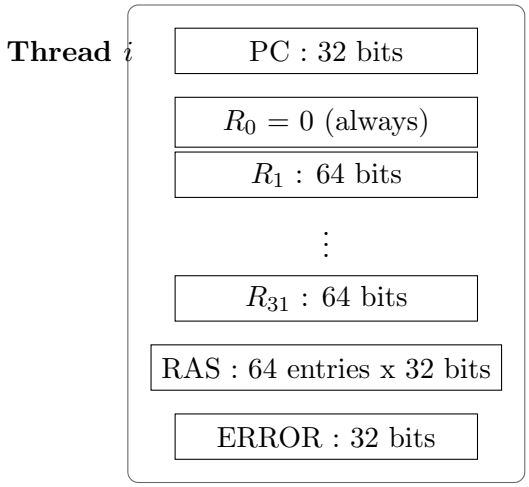


Figure 3.1: Per thread architectural state. The engine maintains four copies of this, one per thread.

- Active: the thread is running. FETCH includes it in its round robin pick.
- Halted: the thread has executed a HALT instruction or has encountered an error condition. FETCH excludes it from its round robin pick. The halted state is sticky until the host resets the thread.

There is no separate "suspended" state. A thread that is waiting on a long latency operation, such as an off island load, simply has its in flight bundle blocked at EXEC. The thread continues to occupy the EXEC slot until the operation completes. The other threads continue to make forward progress through the pipeline.

3.2 Pipeline overview

Figure 3.2 shows the high level data path of the TEU. The five stages communicate through the elastic channels listed in Table 2.4. WRITEBACK can also send a PC correction back to FETCH and a regfile lock release back to REGFILE.

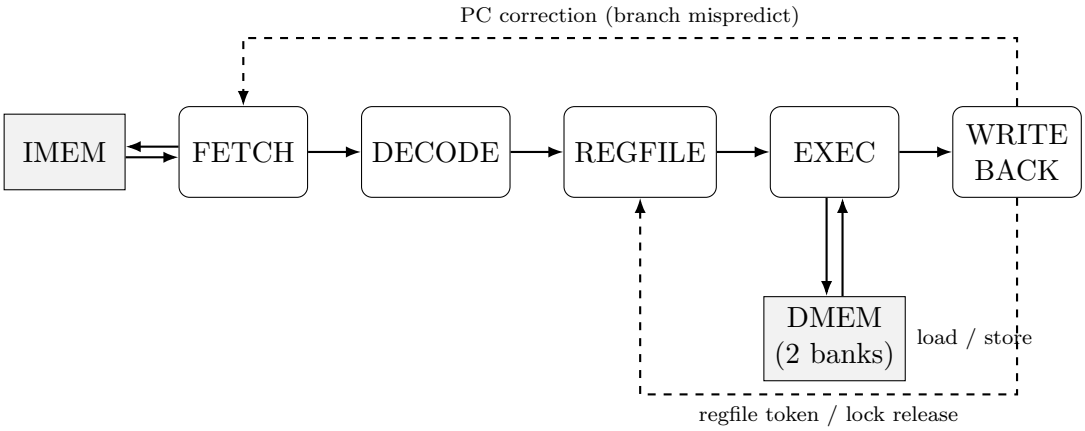


Figure 3.2: TEU pipeline overview. Solid arrows are forward elastic channels. Dashed arrows are feedback paths used for branch mispredict recovery and regfile lock release.

3.3 FETCH stage

FETCH owns the four thread program counters and the branch predictor. It picks an active, not halted thread in round robin order. Halted threads are skipped. With k active threads ($1 \leq k \leq 4$), the pipeline is filled with bundles from those k threads in interleaved fashion.

For each pick, FETCH reads a 256 bit aligned line from IMEM at the address `PC & ~7`. A 256 bit line can hold one or more bundles, since a bundle has 1, 2, 4, or 8 instructions of 32 bits each. For example, a single line may hold a single bundle of 8 instructions, or three bundles of sizes {2, 2, 4}, or two bundles of sizes {4, 4}, and so on. FETCH sequentialises the bundles in the line in PC order. The next bundle from the same line is dispatched before FETCH selects another thread.

[**TO CONFIRM:** We assume that the line to bundle sequentialiser lives inside FETCH and emits one bundle per cycle into a single shared `teu_fetch_to_decode` pipe (depth 2). The hsys does not declare per thread fetch to decode pipes, so the per thread split happens later, at the DECODE to REGFILE boundary.] [**Answer:** Confirmed from `fetch.aa`. The FETCH module contains an internal `fetch_to_issue` pipe (81 bits, depth 4, `$p2p`). The FETCH daemon routes per-thread bundle metadata into this internal pipe; an issue sub-unit inside FETCH reads the 256-bit IMEM line and emits

the padded 336-bit bundle on the single shared `teu_fetch_to_decode` pipe. The per-thread split happens at the DECODE to REGFILE boundary, as assumed.]

3.3.1 NOP padding

The pipeline is sized for a maximum of 8 instructions per bundle. If a bundle has fewer instructions, the unused slots are filled with NOP instructions before the bundle is sent down the pipeline. This keeps the data path width uniform.

[**TO CONFIRM:** It is not yet pinned down whether NOP padding is performed on the way out of FETCH or on entry into the issue/decode boundary. For modelling purposes the difference is not material.] [**Answer:** Confirmed inside FETCH. The internal `fetch_to_issue` pipe carries only 81-bit metadata (bundle start offset within the line, line index, prediction info). The issue sub-unit inside FETCH reads the full 256-bit IMEM line, selects the correct sub-bundle, pads to 8 slots with NOPs, and emits the full 336-bit token on `teu_fetch_to_decode`.]

3.3.2 Branch predictor

[**TO CONFIRM:** The exact replacement and key matching policy needs confirmation. The assumption used here is: PC keyed match, single entry per thread, default not-taken. A PC keyed predictor is necessary to make nested loops behave reasonably.]

[**Answer:** The assumption was wrong in several respects. The correct description from `bpbV3.aa` is given below.]

The branch predictor (`bpbV3`) is a **256-entry direct-mapped shared** branch target buffer (not a per-thread single-entry table). Its structure is:

- Storage: `$array [256] $of $uint<54>`. Each entry holds a 22-bit tag, a 2-bit saturating counter, and a 30-bit predicted target PC word index.
- Index: the lower 8 bits of the PC word index address the table.
- Tag: the upper 22 bits of the PC word index are the match tag.

- Saturating counter: 0 means strongly taken; 3 (=MAX_SAT_COUNT_VAL) means strongly not-taken.

Prediction: a lookup matches if the entry is valid, the 22-bit tag equals the upper 22 bits of the lookup PC, and the counter value is less than MAX_SAT_COUNT_VAL (3). When a match fires, the stored 30-bit target PC is used as the predicted next PC.
Updates: when WRITEBACK sends a PC correction to FETCH, the predictor is updated for the branch PC:

- If the branch was taken: the counter is saturating-decremented toward 0; the target PC is updated.
- If the branch was not taken: the counter is saturating-incremented toward 3.
- For a new entry (no tag match): the counter is set to 0 if the correction was taken, or to 3 if it was not taken.

The predictor is shared across all four threads. Collisions between threads (two threads hashing to the same entry) are resolved by tag mismatch, which simply suppresses the prediction (predict not-taken for that fetch). [**TO CONFIRM:** The interaction between the shared BTB and per-thread correctness under heavy multi-thread usage needs to be confirmed, particularly for nested loops in two threads that alias to the same entry.]

3.4 DECODE stage

DECODE consumes a bundle from `teu_fetch_to_decode`. For each instruction in the bundle, it produces the source register selectors, the destination register selector, the immediate operand if any, and the opcode and type information. It then routes the decoded bundle onto the per thread channel `noblock_teu_decode_to_register_file_read_request_T`, where T is the thread id. Decode itself does not interact with the register file or the architectural state. It is a stateless transformation stage.

3.5 REGFILE stage

REGFILE resolves all read after write hazards. It holds one pending bundle per thread, taken from the four `noblock_teu_decode_to_register_file_read_request` pipes.

[**TO CONFIRM:** A small entry depth per thread queue, assumed at least 2 entries to match the upstream channel depth.] [**Answer:** The Aa_model (`registers_v2.aa`) shows one pending slot per thread (a single `read_command_N` phi variable per thread), not a multi-entry queue. The flow control is via a token counter initialised to 2 (AVAILABLE_TOKENS). At most 2 bundles total, across all threads, can be in flight between REGFILE and WRITEBACK at any time. The upstream noblock pipe (depth 2 from hsys) provides the only upstream buffering.]

3.5.1 Register file locks

The register file maintains a per thread, per register pending write lock count (one bit per register, 32 bits per thread). When a bundle reaches the head of its thread slot, the picker checks whether any source register read by the bundle is locked. If yes, the bundle stays and waits. If no, the bundle's reads are issued, the destination registers are locked (lock bits set), and the bundle proceeds to EXEC. WRITEBACK sends a lock release token that clears the lock bits for the written registers. Until the lock bits are cleared, no later bundle of the same thread can be issued.

The lock is logically per register, but because each thread is strictly in order at REGFILE, the practical effect is that a single locked register blocks every later bundle from the same thread. Bundles from other threads are unaffected. [**TO CONFIRM:** This is the intended behaviour of the strictly in order per thread machine.] [**Answer:** Confirmed from `registers_v2.aa`. `update_register_locks` applies unlock first, then checks test mask and lock mask. A bundle waits if `all_locks_acquired` is false.]

3.5.2 Picker

REGFILE's picker selects one bundle per cycle to send to EXEC. If more than one thread slot has an unblocked bundle, the picker uses round robin. [**TO CONFIRM:** We assume that at most one bundle is issued to EXEC per cycle, so all instructions entering EXEC in the same cycle belong to the same

thread and the same bundle.] [**Answer:** Confirmed from `registers_v2.aa`. `select_read_command` selects exactly one thread slot per cycle; `noblock_teu_register_file_to_exec` is written only when `send_read_response` is true.]

3.5.3 Side ports

REGFILE also services register read and write requests from the CONTROL_UNIT, used by the host to inspect and modify thread state. These are carried on the `noblock_ccu_to_teu_read_register` and `noblock_ccu_to_teu_write_register` pipes. They have lower priority than pipeline traffic and are out of scope for the timing model in the common case.

3.6 EXEC stage

EXEC receives a bundle of up to 8 instructions, all from the same thread, with operand values resolved. It executes all of them in parallel. Each instruction has its own latency that depends on the opcode and the element type (see §4.3). The bundle leaves EXEC only when the slowest instruction in the bundle has finished. While the bundle is in EXEC, no new bundle can enter EXEC.

[**TO CONFIRM:** We assume that EXEC is a single non pipelined stage. It accepts one bundle, holds it for as many cycles as the bundle's slowest instruction needs, then releases it.] [**Answer:** The Aa_model `exec.aa` shows that `exec_daemon` is internally pipelined (`$dopipeline $depth 15 $fullrate`). CORE_DELAY = 3 (from `exec_parameters.aa`) is the pipeline depth used by most operator modules. The pipeline runs at full rate, so a new bundle can enter every cycle; however, the output of each operator module appears after CORE_DELAY cycles. DIV and SQRT use a shared non-replicated unit whose latency is longer. From the perspective of the timing model, the effective bundle latency is the maximum operator latency across all instructions in the bundle.]

3.6.1 Instruction slot alignment constraints

[**New finding from Aa_model:** The Aa_model `exec.aa` enforces hard alignment rules on which instruction slots are legal for each operation class.

A violation raises `instruction_alignment_error`, which immediately halts the thread.]

- **CTI** (branch, call, return): must be placed in slot 0 only.
- **DIV and SQRT**: must be placed in slot 1 only.
- **LOAD and STORE**: must be placed in slots 4, 5, 6, or 7 only. At most two memory operations per bundle (one per DMEM port).
- All other operation classes (ALU, FP, vector, dot, cast, pack, slice, cmov, set, nop, null, halt) may occupy any slot.

These constraints are enforced at hardware level. The assembler tool `mcode_align` is expected to validate slot alignment before assembly.

3.6.2 Memory operations

For load and store instructions (slots 4–7), EXEC issues the request on one of the two DMEM ports. The bundle's effective EXEC latency includes the memory round trip. If the address resolves to a location that is not in the engine's local memory (an off island access), the operation is routed via the DMA engine [**New finding from Aa_model:** (confirmed from `data_mem.aa`: addresses outside the engine's 64 KB local range are handled by `access_external_mem_via_dma`)]. The bundle stays in EXEC for the full external latency. All instructions in the bundle exit EXEC together.

3.6.3 CTI handling and stream id

When a CTI (placed in slot 0) executes, EXEC computes the corrected PC. The corrected PC is forwarded with the bundle to WRITEBACK, which is responsible for sending the PC correction back to FETCH.

Each fetched bundle carries a stream id that identifies the speculative path the bundle was fetched on. [**TO CONFIRM:** We assume the stream id is constructed as (thread_id, epoch), where the epoch counter is incremented per thread on every mispredict recovery. EXEC discards bundles whose stream id does not match the current epoch for its thread.]

[**Answer:** The assumption was wrong. The correct construction from `fetch_core.aa` and `calculate_next_stream_id` is:]

The stream id is 8 bits, structured as two 4-bit epoch counters:

$$\text{stream_id}[7:4] = \text{tid_epoch}, \quad \text{stream_id}[3:0] = \text{sid_epoch}$$

Both are wrapping 4-bit counters, not thread ids.

- **tid_epoch** is incremented each time a thread is newly started or restarted by the CCU (`new_ethread = 1`). It changes when a thread transitions from inactive to active.
- **sid_epoch** is incremented each time a mispredict correction is applied to a running thread (`new_estream = 1`).

Each bundle that FETCH emits carries `new_ethread` and `new_estream` flag bits in its header. EXEC uses these flags, not a direct stream id comparison, to decide whether to run or skip a bundle. The skip logic in `exec.aa` is:

- Skip if the per-thread EXEC state is `EXEC_HALTED` and `new_ethread` is not set.
- Skip if the per-thread EXEC state is `EXEC_MISPREDICTED` and neither `new_ethread` nor `new_estream` is set.

The constants are `EXEC_ACTIVE = 0`, `EXEC_HALTED = 1`, `EXEC_MISPREDICTED = 3`.

3.6.4 Exception detection

[**New finding from Aa_model:** EXEC detects six types of hardware exceptions, one per instruction slot. The exception code field in the ERROR register is 24 bits wide (8 slots × 3 bits each):]

Code	Exception type
001	Integer divide by zero
010	Floating point divide by zero
011	Floating point NaN result from ALU operation
100	Integer square root of a negative number
101	Floating point square root of a negative number
110	Unaligned memory access

When any exception or a halt instruction is encountered, the thread transitions to halted. The exception codes and a slot-level alignment error (`instruction_alignment_error`) are OR-ed into the halt condition.

3.6.5 RAS state and operations

[**New finding from Aa_model:** RAS operations (push on CALL, pop on RETURN) are performed inside EXEC, not in WRITEBACK as one might expect from the pipeline diagram. The Aa_model `exec.aa` maintains the per-thread RAS state in the EXEC circulant as a 52-bit structure per thread: write_pointer (6 bits), read_pointer (6 bits), nitems (8 bits), ras_top_pc (32 bits). The WRITEBACK stage (retire daemon) receives the halt notification from REGFILE and forwards it; it does not itself manage the RAS.]

3.6.6 Vector parallelism

A bundle of up to 8 vector ALU instructions of the same operation class is expected to execute in parallel. [**TO CONFIRM:** This claim is based on the designer's verbal note. It needs to be checked against the hardware: in particular, the case of 8 vector multiplies in a single bundle, which requires a sufficient number of vector multipliers. The same question applies to all vector instructions, not just multiply.] [**Answer:** Confirmed from `exec.aa`. There are 8 independent operation class classifiers (one per slot, `classifyOperationClass`) and 8 independent parallel execution paths for `fp_exec_shared`, `fp_exec_operator`, and `iu_exec`. The FP shared module handles all vector FP operations across 8 slots simultaneously. All 8 instructions in a bundle execute in parallel.]

3.7 WRITEBACK stage

WRITEBACK is the last stage of the pipeline. It receives the bundle's results on `noblock_teu_exec_to_write_back_aggregated`. This pipe is read by REGFILE, which performs register writes and lock releases and then forwards the halt information to the WRITEBACK (retire) daemon.

The retire daemon (`retire.template.aa`) performs two actions:

- it emits a retire trace token on `engine_retire_trace` when tracing is enabled; and

- it forwards halt notifications received from REGFILE on `noblock_teu_halt_pipe` to FETCH.

Note: register writes, lock releases, and RAS updates are all performed inside EXEC and REGFILE, not in the retire daemon. The retire daemon is a lightweight forwarder.

3.7.1 Mispredict signalling

If WRITEBACK detects that the bundle contained a mispredicted CTI, it sends a PC correction on `noblock_teu_pc_correction_pipe`. The correction includes the new PC and the thread id. FETCH consumes this and adjusts the PC for the affected thread. The mispredict is detected only at WRITEBACK. [**TO CONFIRM:** Other stages between FETCH and WRITEBACK do not themselves squash; they may keep processing wrong path bundles up to EXEC. EXEC is the only stage that drops bundles whose stream id has been invalidated.] [**Answer:** Confirmed. EXEC uses the `new_ethread` and `new_estream` flags carried in the bundle header to decide skip vs. run. Pre-EXEC stages pass bundles through without inspection.]

3.7.2 HALT

When WRITEBACK encounters a HALT instruction, it marks the corresponding thread halted by sending on `noblock_teu_halt_pipe`. FETCH then excludes the thread from the round robin from the next cycle. Any further in flight bundles for that thread complete normally; HALT is the last bundle of the thread.

3.7.3 Lock release

WRITEBACK sends a token on `noblock_teu_regfile_to_retire` which decrements the appropriate per register lock counts in REGFILE. Once a register's lock count drops to zero, any waiting reader of that register in the same thread can proceed.

3.8 Memory subsystem

The engine has two disjoint local memories, the data memory (DMEM) and the instruction memory (IMEM). Their architectural sizes were given in Section 1.7; this section describes their microarchitecture — bank organisation, port assignment, bank selection, and off-island routing.

3.8.1 Data memory (DMEM)

The data memory is 64 KB total, organised as two banks of 32 KB each. Each bank has a single 256 bit wide port. The two ports are visible to the TEU as `noblock_ajit_256_wide_bus_request_[0,1]` and `ajit_256_wide_bus_response_[0,1]`.

[**New finding from Aa_model:** Bank selection is determined by a single address bit. From `memory_partitioning.aa`: `bank_id := (a36 [] 15)`, i.e. bit 15 of the 36-bit address selects the bank. The engine's local DMEM base address is $0x100000 + (\text{cluster} \times 0x40000) + (\text{engine} \times 0x10000)$, with a total span of 64 KB per engine (bits [15:0] are the intra-engine offset; bit [15] selects the bank).]

A bundle can issue at most two memory operations (slots 4–7) because there are only two ports, one per bank. Each operation is steered to the bank determined by bit 15 of the computed address. If two memory operations in the same bundle target different banks, they proceed in parallel. If they target the same bank, the requests are serialised. The decision is made at runtime, since memory addresses are computed in EXEC and cannot be checked at compile time.

[**TO CONFIRM:** The exact serialisation rule is: same bank conflicts are queued at the bank, smallest source instruction position first. The bundle's effective EXEC latency includes both serialised accesses.] [**Answer:** The bank selection rule (bit 15) is confirmed. The serialisation order (smallest slot position first) is not explicitly stated in the Aa_model source files scanned. For the Sitar model we will apply the stated rule (smallest slot index first), but note that the Aa_model hardware description does not enforce a deterministic serialisation order for same-bank conflicts, which will make direct comparison between the Sitar model and the Aa_model non-deterministic in those cases. **The Aa_model should be updated to enforce the same serialisation order, so that both models produce identical memory access sequences and can be validated against each other.** This is

flagged as an action item for Prof. Desai.]

3.8.2 Instruction memory (IMEM)

The instruction memory is 16 KB. [**TO CONFIRM:** The current C reference model declares the instruction memory as `uint32_t[2048]`, which is 8 KB. The header comment and chapter 1 of the AparaReferenceV2 document state 16 KB. We adopt 16 KB here per the designer.] [**Answer:** Confirmed 16 KB from `imem.aa`: the bank address field is 9 bits wide (512 lines $\times$ 256 bits/line = 512 $\times$ 32 bytes = 16 KB). The C model's `uint32_t[2048]` (8 KB) remains a known discrepancy.]

Chapter 4

Timing

4.1 Cycle model

Time advances in clock cycles. Each elastic channel is a FIFO with the depth declared in the corresponding hsys file. A write to a full FIFO stalls the writer. A read from an empty noblock FIFO returns immediately with valid clear. The reader can do other work in that cycle. The FIFO depths are listed in Table 2.4.

4.2 Per stage cycle behaviour

Table 4.1 summarises the cycle level behaviour of each pipeline stage.

4.3 Per opcode latency

The latency of each opcode is configurable in the timing simulator and is loaded at startup from a configuration file.

[**TO CONFIRM:** The 3 cycle baseline for nearly all operations and the 10 cycle figure for divide and square root are designer's verbal values. They need to be cross checked against the RTL.] [**Answer:** The 3-cycle baseline is not correct for all operations. The Aa_model `docs/OperatorClassification.txt` gives a type-dependent table: latency is 1, 2, or 3 cycles depending on the operand element type, as detailed in Table 4.2 below. CORE_DELAY = 3 (from `exec_parameters.aa`) is

Table 4.1: Cycle level behaviour of each pipeline stage.

stage	cycle behaviour
FETCH	one IMEM line fetch per cycle (when a thread is picked); one bundle dispatched to DECODE per cycle from the line buffer, in PC order
DECODE	one bundle decoded per cycle, routed to one of four per thread channels
REGFILE	one bundle picked into EXEC per cycle, subject to RAW lock checks; same cycle accepts a lock release from WRITEBACK
EXEC	holds one bundle from arrival until the slowest instruction in the bundle finishes; latency is the maximum opcode latency in the bundle, plus memory round trip if any of the bundle's instructions is a load or store
WRITEBACK	one bundle retired per cycle; sends PC correction on mispredict, halt notification on HALT, and lock release token to REGFILE

the FP pipeline depth and applies only to wide-type FP operations. The 10-cycle figure for DIV and SQRT was not found in the Aa_model source; it is retained as the verbal value and flagged for confirmation.]

Table 4.2 gives the per-opcode-class latency. *Element type* refers to the scalar or vector element type of the operands. Opcodes marked with † appear in `parameters.aa` (the assembler opcode table) but not in `OperatorClassification.txt`; their latencies are to be confirmed.

4.4 Stall and squash behaviour

- RAW stall: if a bundle reads a register whose lock bit is set, the bundle waits at its REGFILE slot. The stall length equals the time from slot arrival until the pending writer's WRITEBACK clears the lock.
- Bank conflict: same bank memory operations in the same bundle serialise at the bank. The bundle's EXEC latency increases by the serialised access time. Serialisation order: smallest source instruction slot index first (assumed in Sitar model; Aa_model currently non-deterministic for same-bank conflicts, see §3.8).
- Off island access: the bundle's EXEC latency equals the external transaction latency. All other instructions in the same bundle wait for it.

- Branch mispredict: bundles fetched on the wrong path occupy pipeline slots up to EXEC. They are skipped at EXEC entry once EXEC enters MISPREDICTED state for that thread. The penalty is the time from mispredict detection at WRITEBACK to the corrected fetch landing at EXEC. With the FIFO depths in Table 2.4, this is on the order of 4 to 6 cycles for a single active thread; with multiple active threads the penalty is masked because other threads keep EXEC busy.

- HALT: a thread that halts vacates its slot in FETCH's round robin. Other threads continue. The pipeline does not flush; the halted thread's already issued bundles complete normally.

Table 4.2: Per opcode class latency by element type. Values are in clock cycles.

Opcode class	Element type	Cycles	Notes
NOP, NULL	any	1	
BRANCH, CALL, RET	any	1	CTI must be in slot 0
HALT	any	1	
SET, SLICE, PACK	any	1	
OR, AND, XOR, NOR, NAND, XNOR	int (any)	1	
ADD, SUB, MUL	int $\leq$ 32-bit; fp $\leq$ fp16	1	
ADD, SUB, MUL	int 64-bit	2	
ADD, SUB, MUL	fp32, fp64	3	
SHL, SHR	int $\leq$ 32-bit	1	
SHL, SHR	int 64-bit	2	
DOT	int $\leq$ 32-bit; fp $\leq$ fp32	2	
DOT	fp32, fp64, int64	3	
CAST	int$\rightarrow$int; fp$\rightarrow$int; fp$\rightarrow$fp $\leq$ fp16	1	
CAST	fp32+; fp-fp32/64	2	
LOAD, STORE (local)	all	3	plus same-bank serialisation penalty if applicable
LOAD, STORE (off-island)	all	arbitrary	blocks bundle in EXEC until DMA returns
VADD, VSUB, VMUL, VSCALE	same as scalar ADD/SUB/MUL	1/2/3	type-dependent, same rules as scalar
VREDUCE†	—	[a]	
CMOV†	—	[a]	
DIV	all	10[b]	single shared divider unit
FSQRT	all	10[b]	single shared FP unit

[a] **[TO CONFIRM:** not in OperatorClassification.txt] [b] verbal value; **[TO CONFIRM:** not found in Aa_model, to be confirmed]

Chapter 5

Opcode Numbering Discrepancy

[**TO CONFIRM:** The Aa_model contains two files with opcode number assignments that partially conflict for opcodes 0x16 and above.]

Table 5.1 shows the discrepancy. `parameters.aa` (in the Aa_model) and `engine_isp/isa/parameters.aa` (used by the assembler and McodeMachine) agree on all opcodes 0x00 through 0x15. From 0x16 onwards they differ. `OperatorClassification.txt` (also in the Aa_model docs) uses the second numbering but contains no VREDUCE or CMOV entries.

Table 5.1: Opcode numbering from 0x16 onwards: two sources compared.

Opcode	`parameters.aa` (assembler / McodeMachine)	`OperatorClassification.txt`
0x16	VREDUCE	VADD
0x17	CMOV	VSUB
0x18	PACK	VMUL
0x19	SLICE	VSCALE
0x1a	NULL	SLICE
0x1b	HALT	PACK
0x1c	—	NULL
0x1d	—	HALT
	VREDUCE and CMOV present VADD/VSUB/VMUL/VSCALE absent	VREDUCE and CMOV absent VADD/VSUB/VMUL/VSCALE present

Action required from Prof. Desai: which of the two opcode tables reflects the current hardware and Aa_model? In particular: (1) are VADD, VSUB, VMUL, VSCALE distinct opcodes or are they the same opcode as VALU with type-field variants? (2) what are the correct latencies for VREDUCE and CMOV? (3) is `OperatorClassification.txt` from an older version of the ISA?

This discrepancy must be resolved before the Sitar model can encode the correct opcode-to-latency mapping.

Chapter 6

Open Items

The following items are flagged in the body of the document. Items resolved by the Aa_model scan are marked [**Resolved**]. Items still open are marked [**Open**].

1. [**Resolved**] ERROR register layout. Bits [31:27] unused; [26] ras-error; [25] halt-flag; [24] exception-flag; [23:0] exception-codes (8 slots × 3 bits). See §2.4 and Table 2.5.

2. [**Resolved**] ERROR register host visibility. Accessible via CCU register [5] through the AFB read-register path.

3. [**Resolved**] Issue/sequentialiser inside FETCH. Confirmed. Internal `fetch_to_issue` pipe (81 bits, depth 4) inside `fetch.aa`. The issue sub-unit expands the IMEM line into a padded 336-bit bundle before emitting on `teu_fetch_to_decode`.

4. [**Resolved**] NOP padding location. Confirmed inside FETCH's issue sub-unit.

5. [**Resolved, original assumption wrong**] Branch predictor. 256-entry shared direct-mapped BTB (`bpbV3.aa`), 2-bit saturating counter, 22-bit tag + 8-bit index from PC word index. Full description in §3.3.2. [**Open**] Effect of cross-thread collisions in the shared BTB on prediction quality for nested loops.

6. [**Resolved, original assumption wrong**] Stream id construction. 8-bit value = tid_epoch (4 bits) —— sid_epoch (4 bits). EXEC uses

`new_ethread`/`new_estream` flags, not a raw epoch comparison. Full description in §3.6.3.

7. [**Resolved**] Squash policy. EXEC maintains per-thread `exec_state` and skips bundles using the flag-based logic described in §3.6.3.

8. [**Resolved, original assumption wrong**] REGFILE per thread queue depth. One slot per thread plus 2-token flow control. See §3 REGFILE section.

9. [**Resolved**] Single bundle per cycle into EXEC. Confirmed.

10. [**Resolved, clarified**] EXEC internal structure. Internally pipelined (depth 15); CORE_DELAY = 3 for most ops. See §3 EXEC section.

11. [**Resolved**] Vector instruction parallelism. All 8 slots execute in parallel. See §3 EXEC.

12. [**Resolved, original assumption wrong**] Per opcode latency. Not flat 3 cycles; 1/2/3 cycles depending on element type. See Table 4.2. [**Open**] VREDUCE and CMOV latencies not in OperatorClassification.txt.

13. [**Partially resolved**] DMEM same bank serialisation. Bank selection by address bit 15 confirmed. Serialisation order (slot index, smallest first) not confirmed from Aa_model; to be applied in Sitar model and Aa_model must be updated to match for deterministic cross-model validation.

14. [**Resolved**] IMEM size: 16 KB confirmed from `imem.aa`.

15. [**Open, new**] DIV and FSQRT latency. Verbal value 10 cycles retained; not found in any Aa_model source file. Confirmed as single-shared units from `OperatorClassification.txt`. Need RTL or designer confirmation.

16. [**Open, new**] Opcode numbering discrepancy for opcodes $\geq$ 0x16. See Chapter 5. Must be resolved before Sitar model can encode the correct opcode-to-latency mapping.

17. [**Open, new**] VREDUCE and CMOV latencies. Not present in `OperatorClassification.txt`. Need designer input.

18. [**Open, new**] Shared BTB cross-thread collision behaviour. A 256-entry table shared across all 4 threads may have aliasing between threads that hash to the same entry. Needs confirmation that this is acceptable or intentional.

19. [**Open, new**] Aa_model same-bank conflict determinism. The Aa_model `data_mem.aa` does not enforce a deterministic serialisation order for same-bank memory conflicts within a bundle. This must be fixed in the Aa_model before cross-model validation can be performed.