picoJava™-II Microarchitecture
March 1999

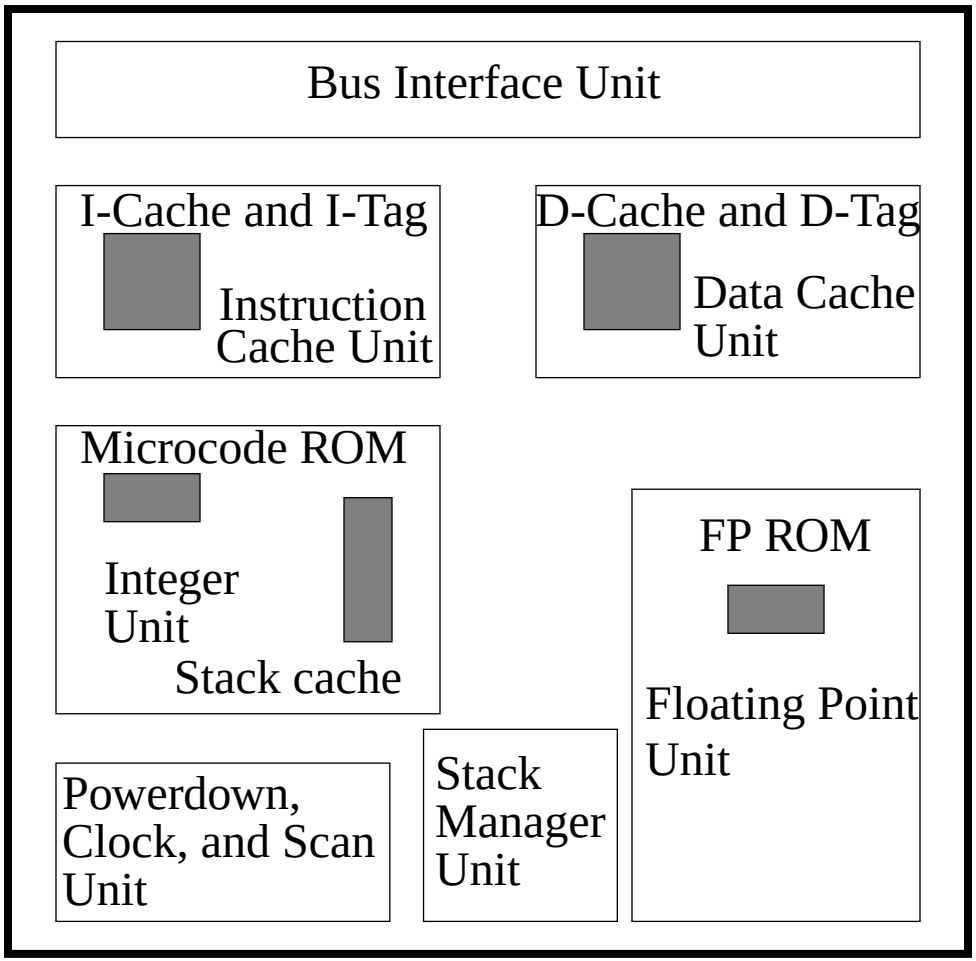
THE NETWORK IS THE COMPUTER®

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300

Features

Feature	**Description**
Pipeline	Six stages
Folding	Up to four instructions
I-Cache line size	16 bytes
I-Cache bus to I-Buffer	64 bits
I-Buffer entries	16
Multicycle operations	Microcode

Core Diagram



Pipeline

Fetch	**Decode**	**Register**	**Execute**	**Cache**	**Writeback**
F	D	R	E	C	W

- **Fetch (F) stage** — The ICU fetches instructions from either the instruction cache in the ICU or from external memory.
- **Decode (D) stage** — The IU groups and precodes instructions in the Instruction Folding Unit (IFU).
- **Register (R) stage** — The IU fetches the operands from the stack cache and determines load-use conditions, bypass conditions, and stack cache miss conditions. The logic for this stage is in the Register Control Unit (RCU).
- **Execution (E) stage** — The IU uses the ALU to either compute arithmetic or calculate the address of a load or store from the DCU. All multicycle operations use the microcode datapath for execution.

Pipeline (Cont’d)

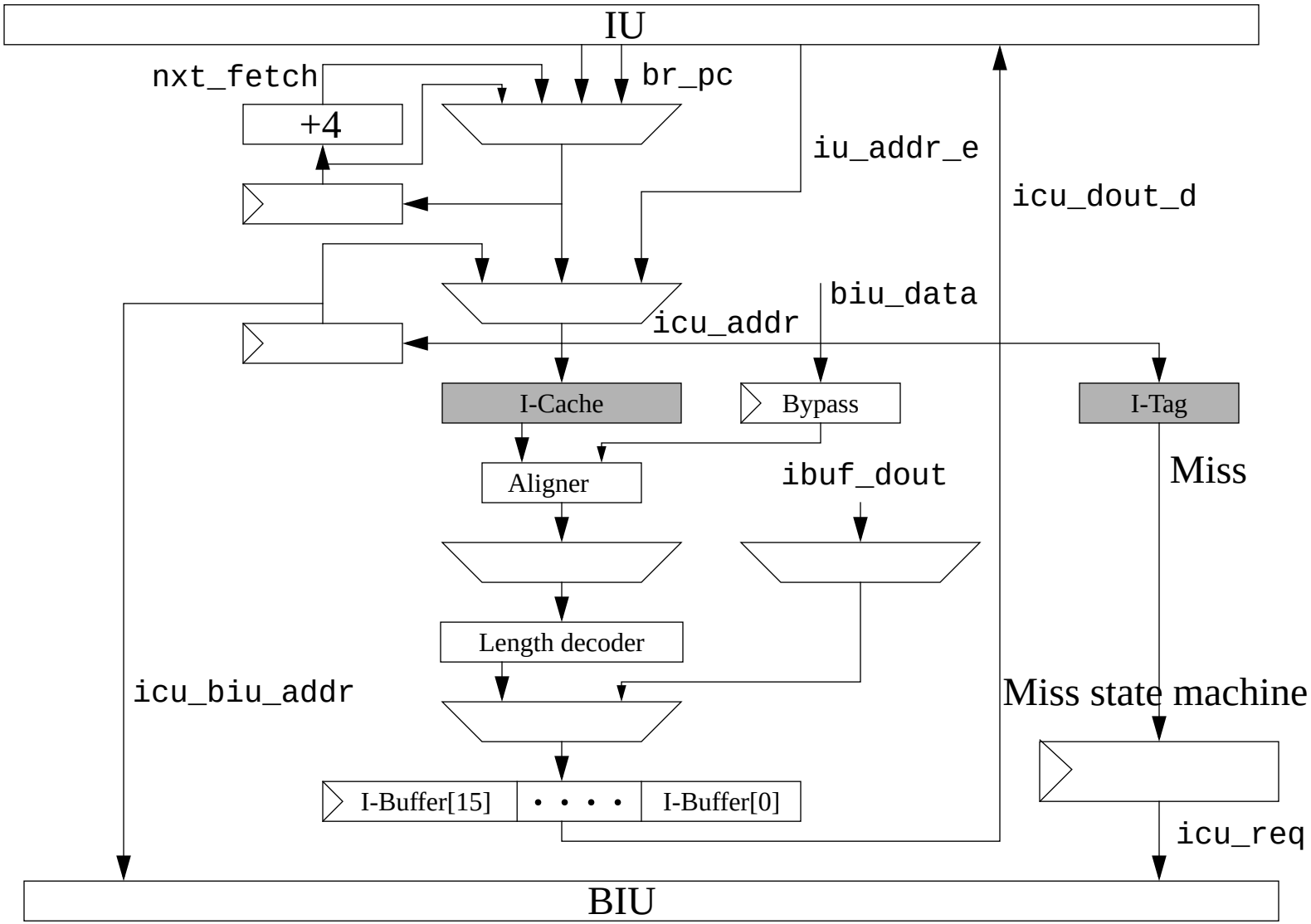
  - In case of a control flow instruction, the IU calculates the branch address and the condition upon which the branch is dependent.
  - In case of a floating-point instruction, the IU provides the operands to the FPU for execution.

- **Cache (C) stage** — The IU accesses data from the data cache, prioritizes, and takes traps at the end of the cycle.
- **Write (W) stage** — The IU writes back the results to the stack cache.

Instruction Cache Unit (ICU)

- **The ICU fetches instructions from the I-Cache and provides them to the decode unit.**
- **The ICU uses an instruction buffer (I-Buffer) to hold instructions that are fetched from memory until they are consumed by the IU.**
- **The I-Buffer gets 8 bytes from the I-Cache or 4 bytes from the Bus Interface Unit (BIU).**
- **The instruction cache line is 16 bytes.**
- **The I-Cache is configurable between 0 to 16 Kbytes.**
- **In a cache miss, the ICU generates a memory request for the missed line.**
- **If the I-Buffer is empty and an instruction cache miss occurs, the decode unit is stalled.**

ICU Interaction with Other Units

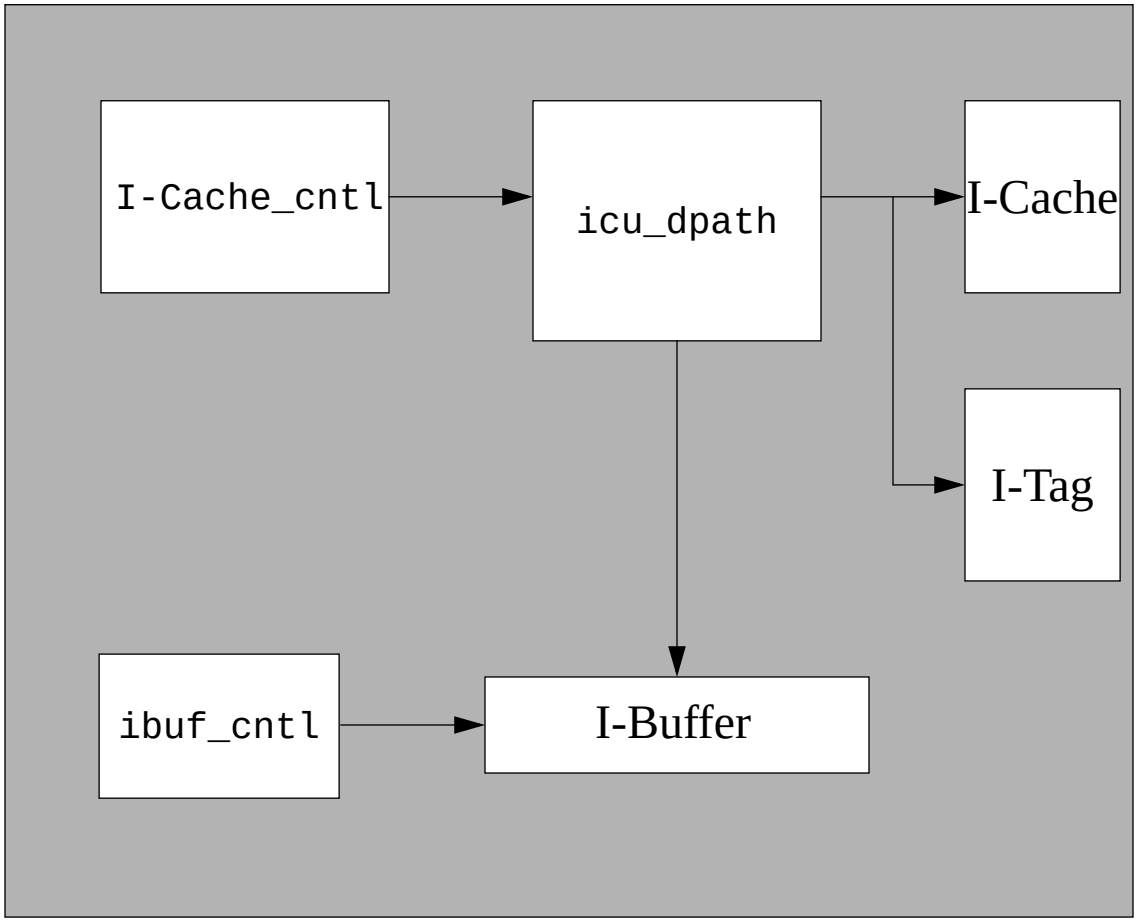


ICU Structure

- **The I-Cache is a direct-mapped, 16-byte line size cache with a single-cycle latency.**
- **Each line has an 18-bit cache tag entry and an associated valid bit.**
- **A 16-byte instruction buffer delinks the fetch stage from the rest of the pipeline for performance reasons.**
- **The processor status register contains an Instruction Cache Enable (`ICE`) bit, which, if disabled, causes instruction fetches to behave as if they had missed in the instruction cache.**

  In that case, the ICU fetches instructions from memory and forwards them to the I-Buffer, but does not write them into the instruction cache.

ICU Structure



ICU Structure (Cont'd)

- **Instruction cache control (`ic_cntl`)** — `ic_cntl` determines which instruction in the instruction cache to access. Depending on the situation, it uses the branch target `PC`, the trap `PC`, or the next `PC` for the access.

  `ic_cntl` also provides mux control for the data mux, which selects between the cache fill bus and the diagnostic data.

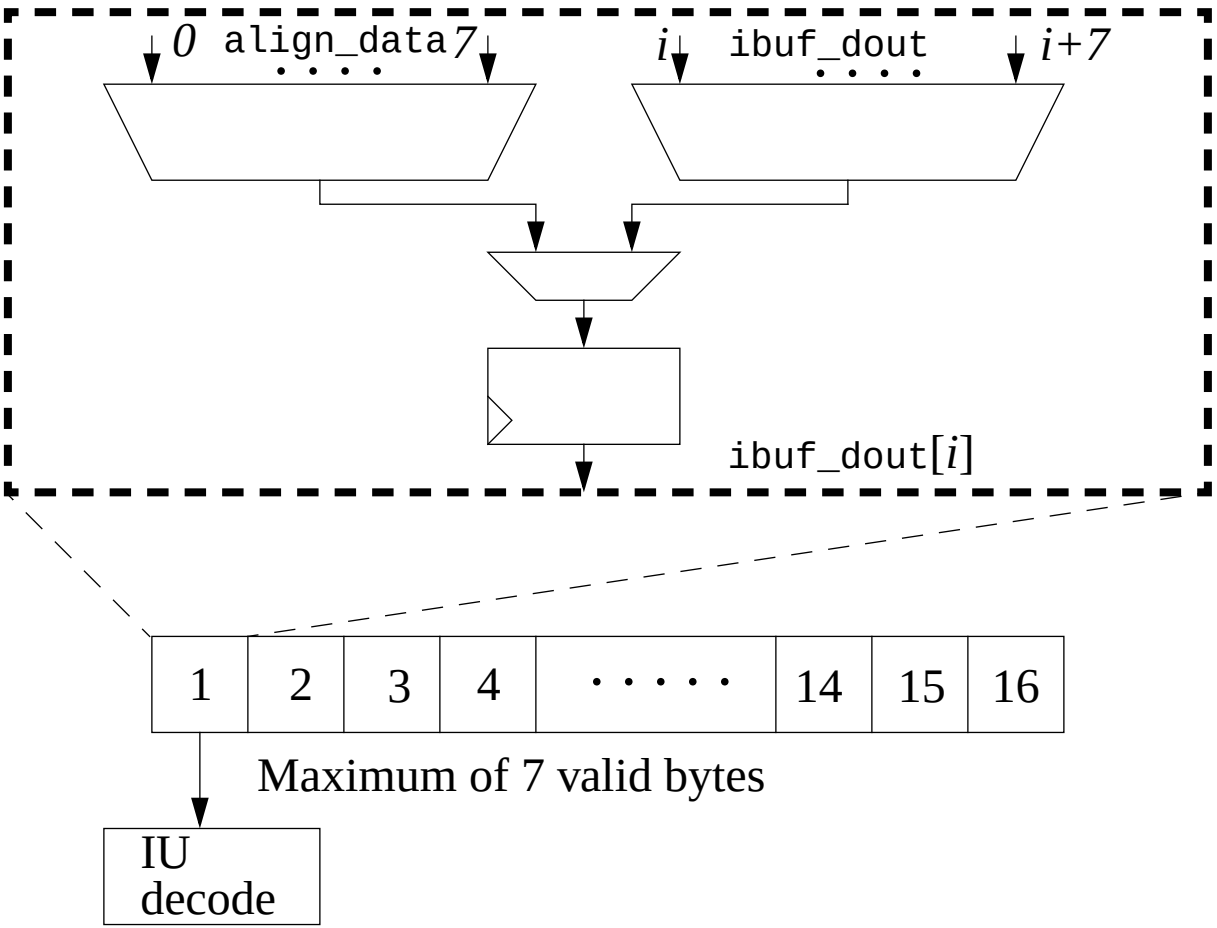
  `ic_cntl` generates bus requests due to cache misses or noncacheable requests. On a cache miss, `ic_cntl` bypasses data from the ICU to the I-Buffer as soon as data are available from memory prior to the completion of the cache fill.

- **Instruction buffer (I-Buffer)** — The I-Buffer is implemented as a 16-byte deep FIFO buffer.

  In a cycle, the ICU adds 8 valid bytes to the I-Buffer from the I-Cache, starting from the first available position. When the I-Buffer is full, the fill transaction stops. The IU can read a maximum of 7 valid bytes, starting from position one.

  When a branch or trap occurs, the ICU flushes all the entries in the I-Buffer, causing the branch or trap data to move to the top of the I-Buffer.

ICU Structure (Cont'd)



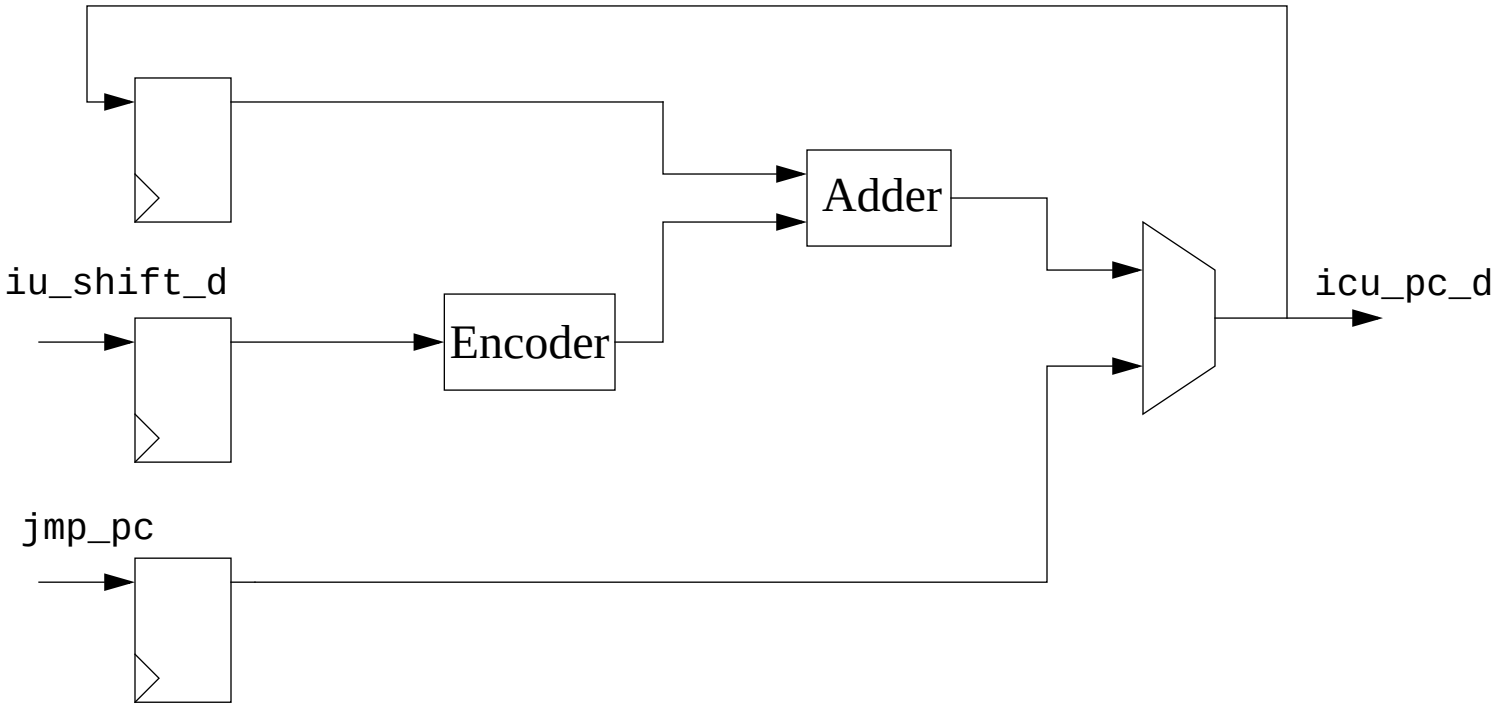
ICU Structure (Cont'd)

- **I-Buffer control (`ibuf_cntl`)** — I-Buffer control provides mux selects and other control signals for I-Buffer functions. It keeps track of the valid bits.

  To simplify the I-Buffer full logic and enhance the timing, the I-Buffer is full if the valid entries exceed 8 bytes.

- **Instruction cache datapath (`icu_dpath`)** — `icu_dpath` consists of the address and the datapath logic. It includes the datapath for generating the `PC` of the next instruction that is to be decoded and does the following:
  - Provides appropriate addresses and data to the RAMs
  - Stores valid entries on to the I-Buffer
  - Supports diagnostic reads and writes to the cache

ICU Structure (Cont'd)



PC Datapath

I-Cache Transactions

- **Cache read hits** — On a read hit, the ICU fetches 8 bytes of data from the instruction cache and stores them in the I-Buffer in one cycle.

  ICU decodes the instruction-length information associated with every byte of the data from the instruction cache into 4 bits per byte and writes them onto the length entry of the I-Buffer.

- **Cache read misses** — On a cache miss, miss control generates a cache fill request to the BIU.

  On receiving the data from memory, the ICU writes the data into the cache and bypasses them into the I-Buffer.

- **Noncacheable reads** — The ICU sends a noncacheable request to the BIU for non-cacheable instructions.

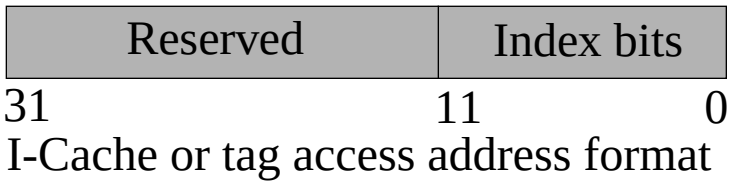
  Once the data are available on the cache fill bus, the ICU bypasses them into the I-Buffer and does not write them into the I-Cache.

I-Cache Transactions (Cont'd)

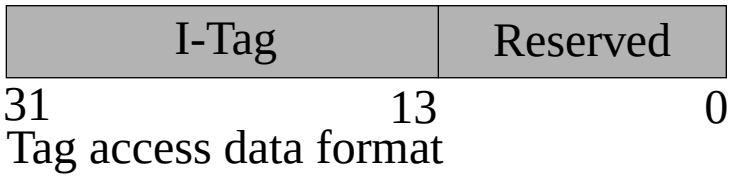
- **Cache indexed flushing** — The ICU can invalidate an instruction cache line without comparing its tag against the address provided by the IU—a protocol that is helpful in self-modifying code.

  During a flush, the ICU accesses the tag for the indexed line and resets the valid bit.

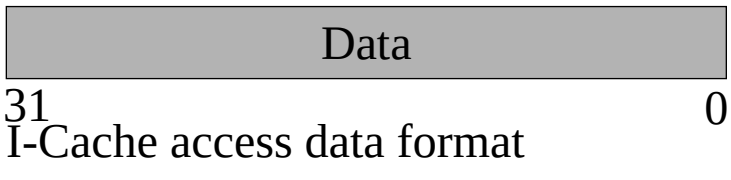
- **Diagnostic reads and writes** — The ICU can use privileged instructions to perform diagnostic reads and writes to both the instruction cache RAM and the I-Tags



I-Cache or tag access address format



Tag access data format



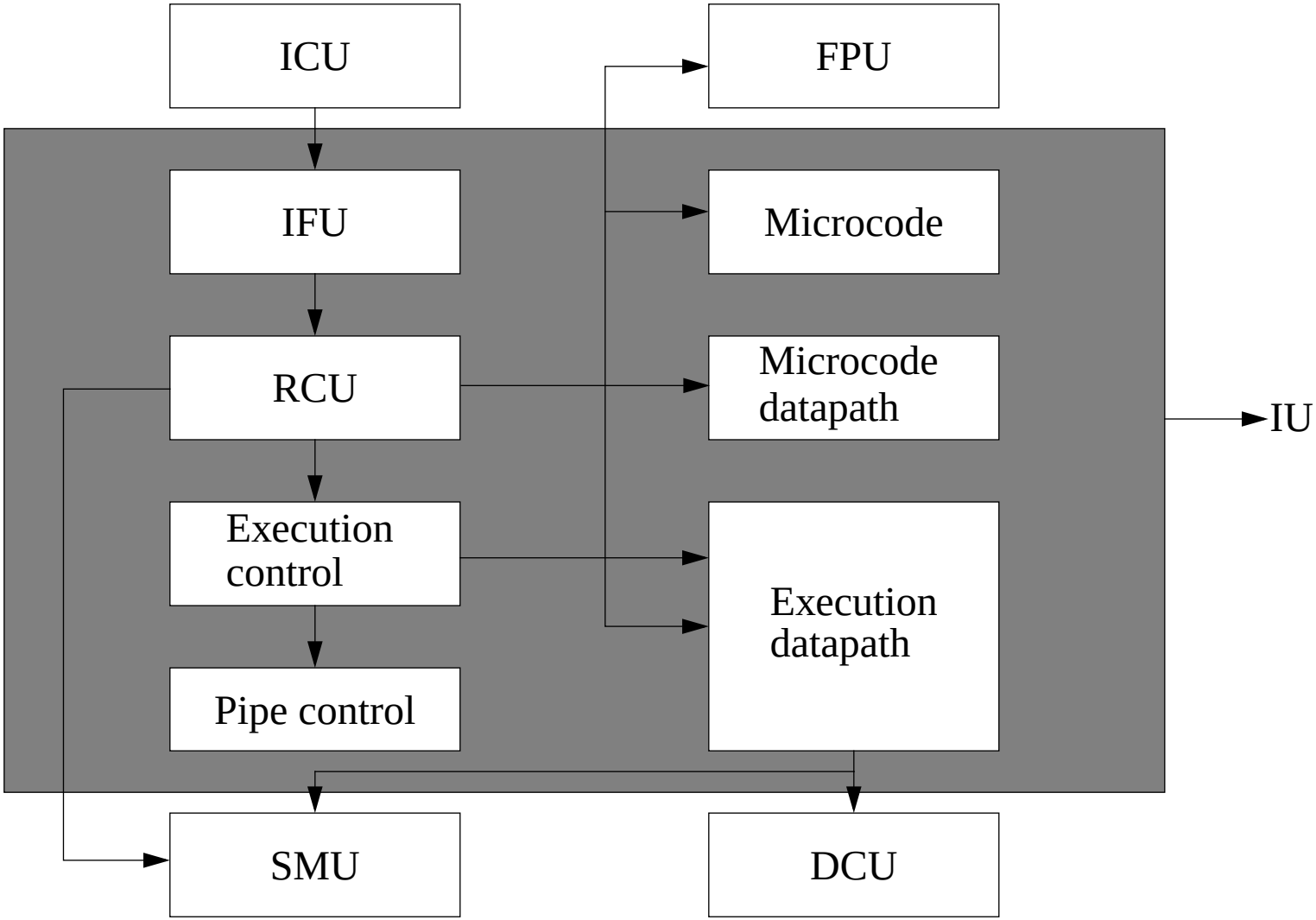
I-Cache access data format

Integer Unit (IU)

The Integer Unit (IU) does the following:

- **Executes all the instructions that are defined in the *picoJava-II Microarchitecture Guide***
- **Interacts with the ICU to fetch instructions**
- **Interacts with the FPU to execute floating-point instructions**
- **Interacts with the DCU to execute load-related and store-related instructions**
- **Groups the instructions in the Instruction Folding Unit (IFU)**
- **Accesses registers and provides bypass signals in the Register Control Unit (RCU)**
- **Decodes the instruction in the Decode Unit and executes the instruction in the execution and microcode datapaths**

Integer Unit (IU) (Cont'd)



Instruction Folding Unit (IFU)

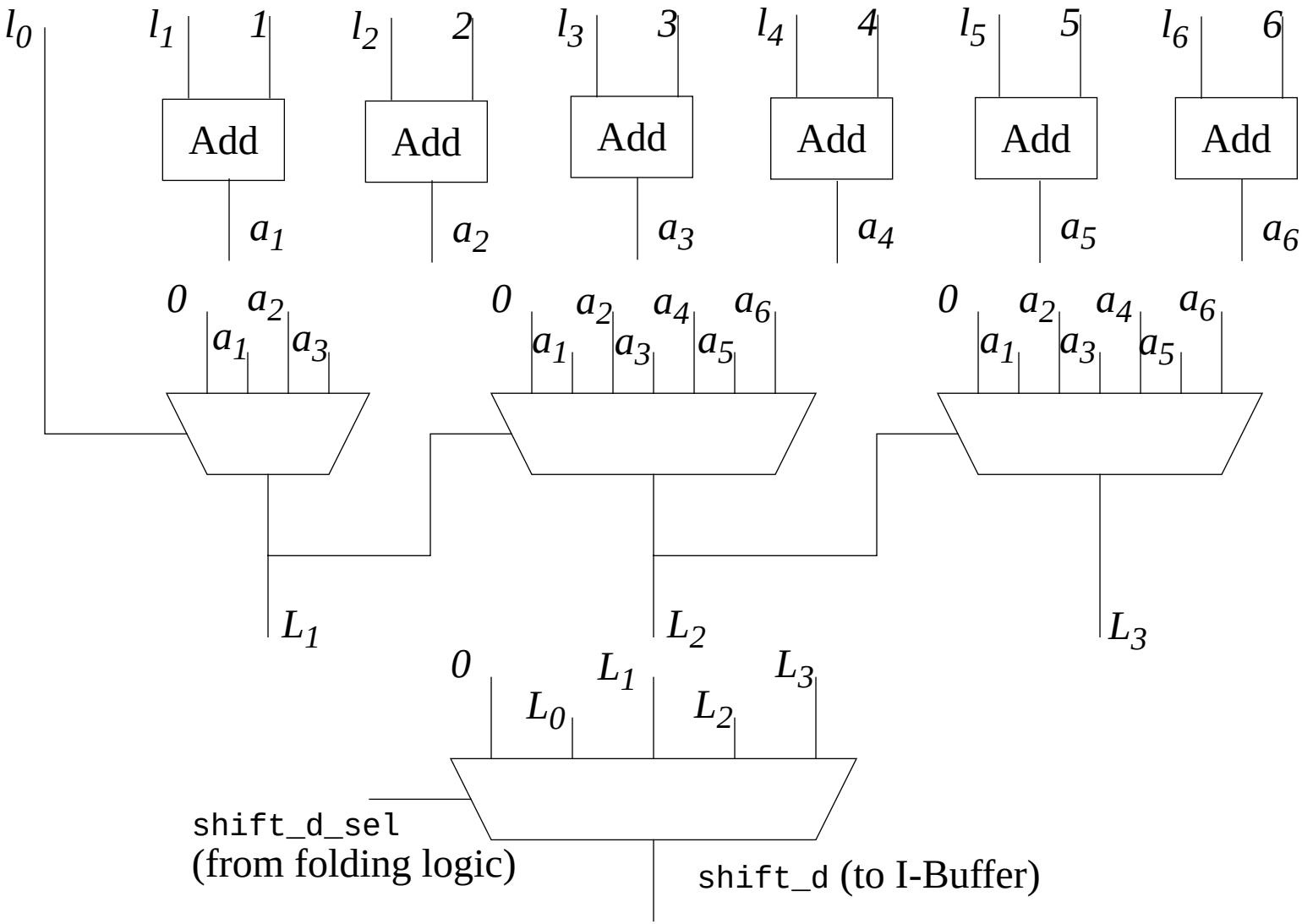
The IFU does the following:

- **Examines the top 7 bytes in the I-Buffer to determine how many instructions can be folded (up to a maximum of four). (See “Folding Groups” on page 23.)**
- **Redecodes the instructions and provides the information to the R stage and the shift signal to the I-Buffer that indicates the number of bytes consumed.**

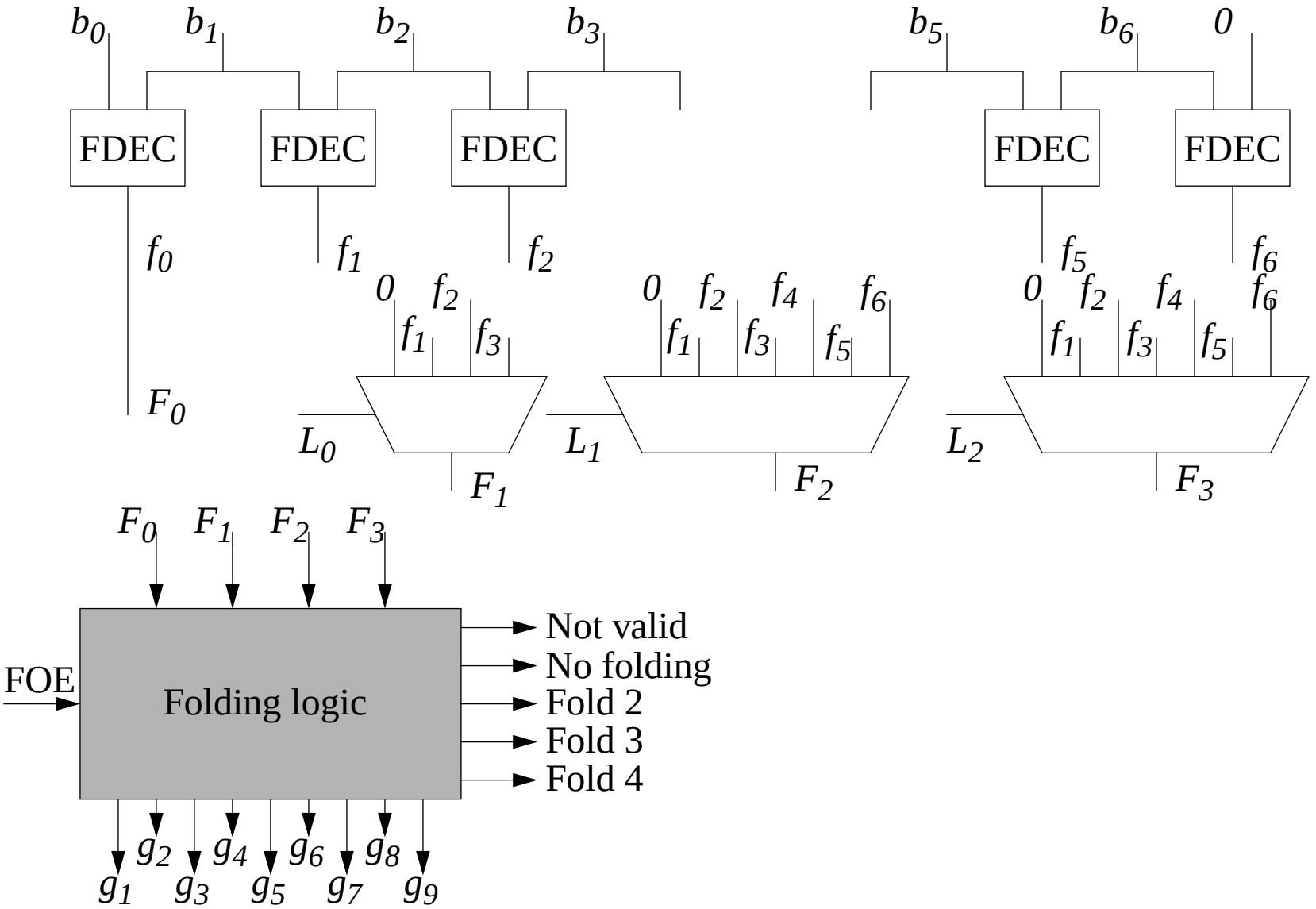
Each of the 16 I-Buffer entries, b_i, contains:

- v_i: Valid bit
- d_i: Dirty bit
- 4-bit l_i: The length of the instruction that corresponds to the opcode b_i

Length Decoder



Folding Decoder and Folding Logic



Instruction Type

- **LV** — Local variable load or load from global register or push constant
- **OP** — Operation that uses the top two entries of the stack. It can be followed by a MEM group instruction.
- **BG2** — Operation that uses the top two entries of the stack and breaks the group
- **BG1** — Operation that uses only the top most entry of the stack and breaks the group
- **MEM** — Local variable stores and global register stores
- **NF** — A nonfoldable instruction

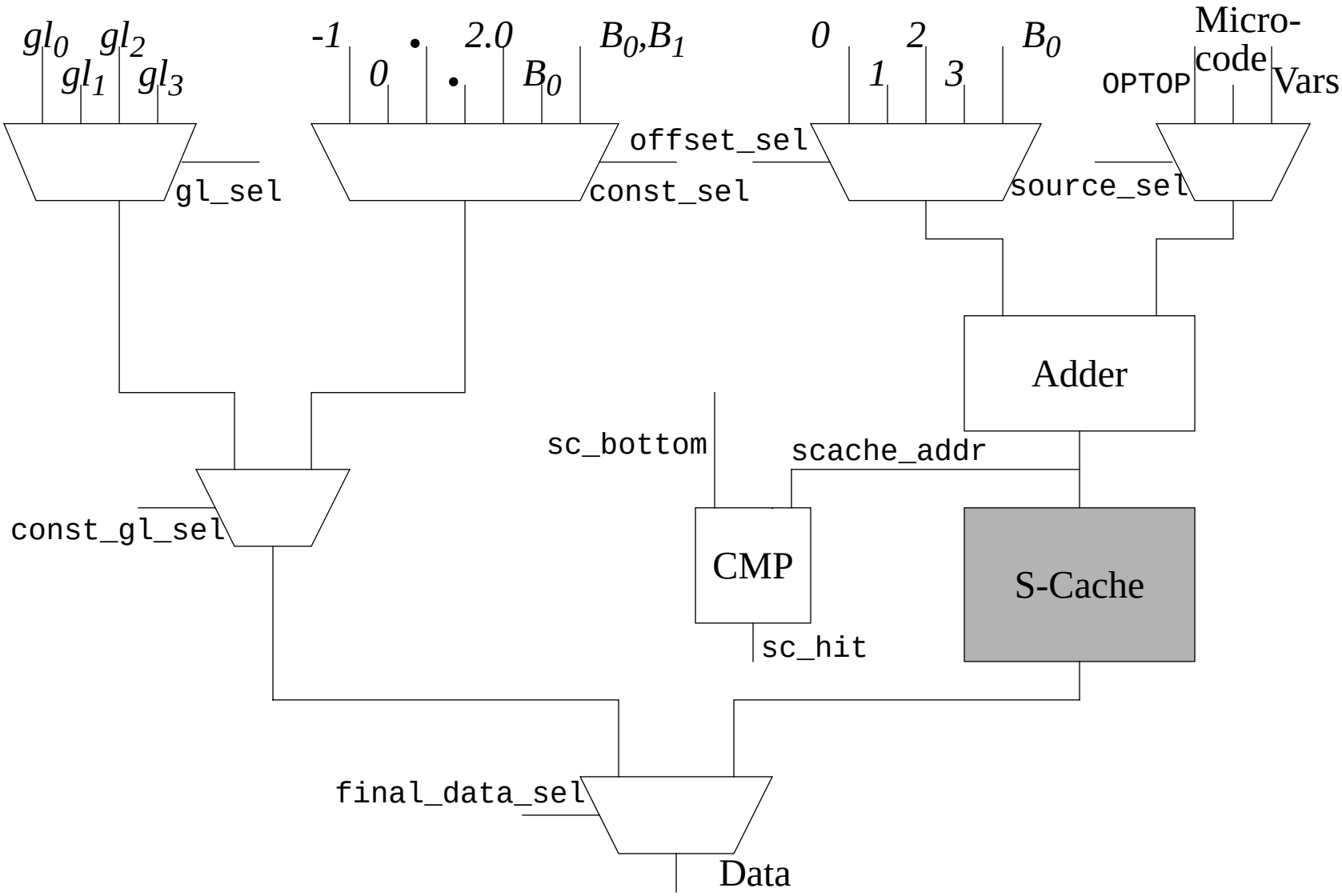
Folding Groups

- **LV LV OP MEM**
- **LV LV OP**
- **LV LV BG2**
- **LV OP MEM**
- **LV BG2**
- **LV BG1**
- **LV OP**
- **LV MEM**
- **OP MEM**

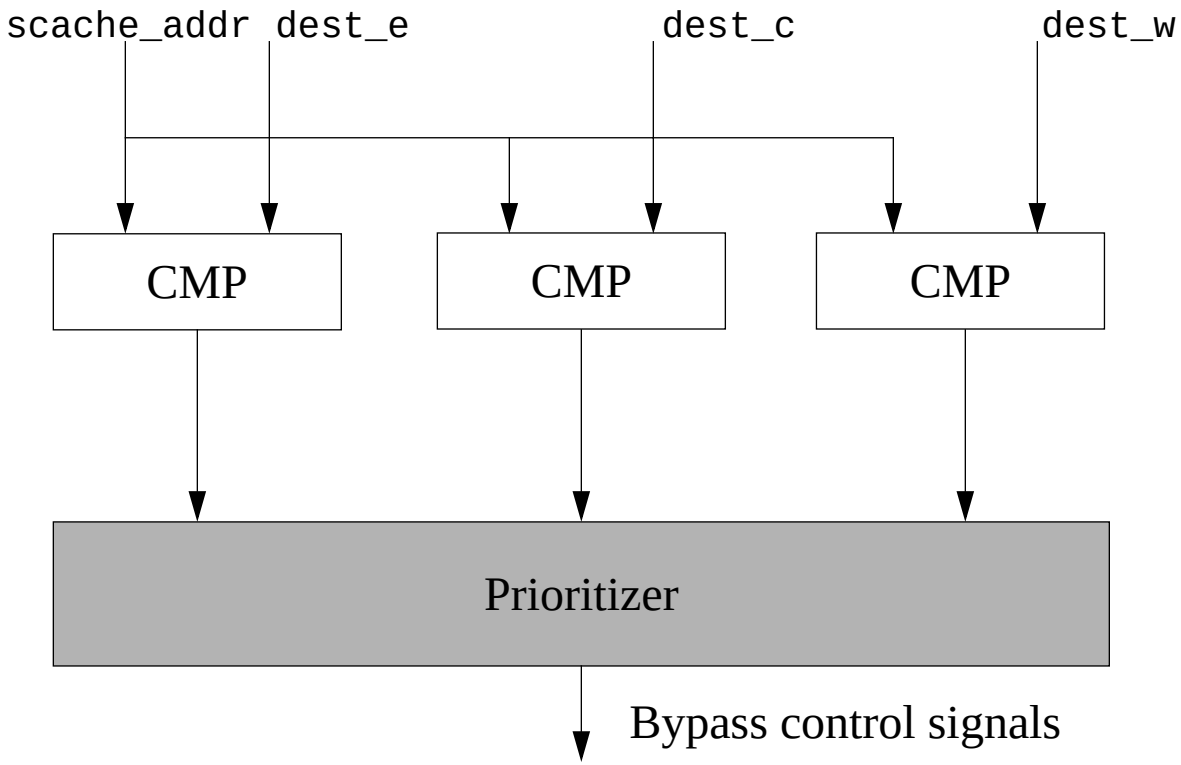
Register Control Unit (RCU)

- **The RCU is composed of the register access logic, the bypass logic, and the destination logic.**
- **The register access logic is responsible for:**
  - Generating constants
  - Fetching operands from the stack
  - Determining stack cache hits or misses
- **The bypass logic does the following:**
  - Determines the operand bypass controls, which bypass data from older stages in the E stage of the pipe
  - Generates the load use case, during which the DR stage of the pipe is stalled
- **The destination logic does the following:**
  - Determines the destination address for each operation in the R stage
  - Keeps track of the destination address down the pipe
  - Writes to the stack cache in case of cache hits

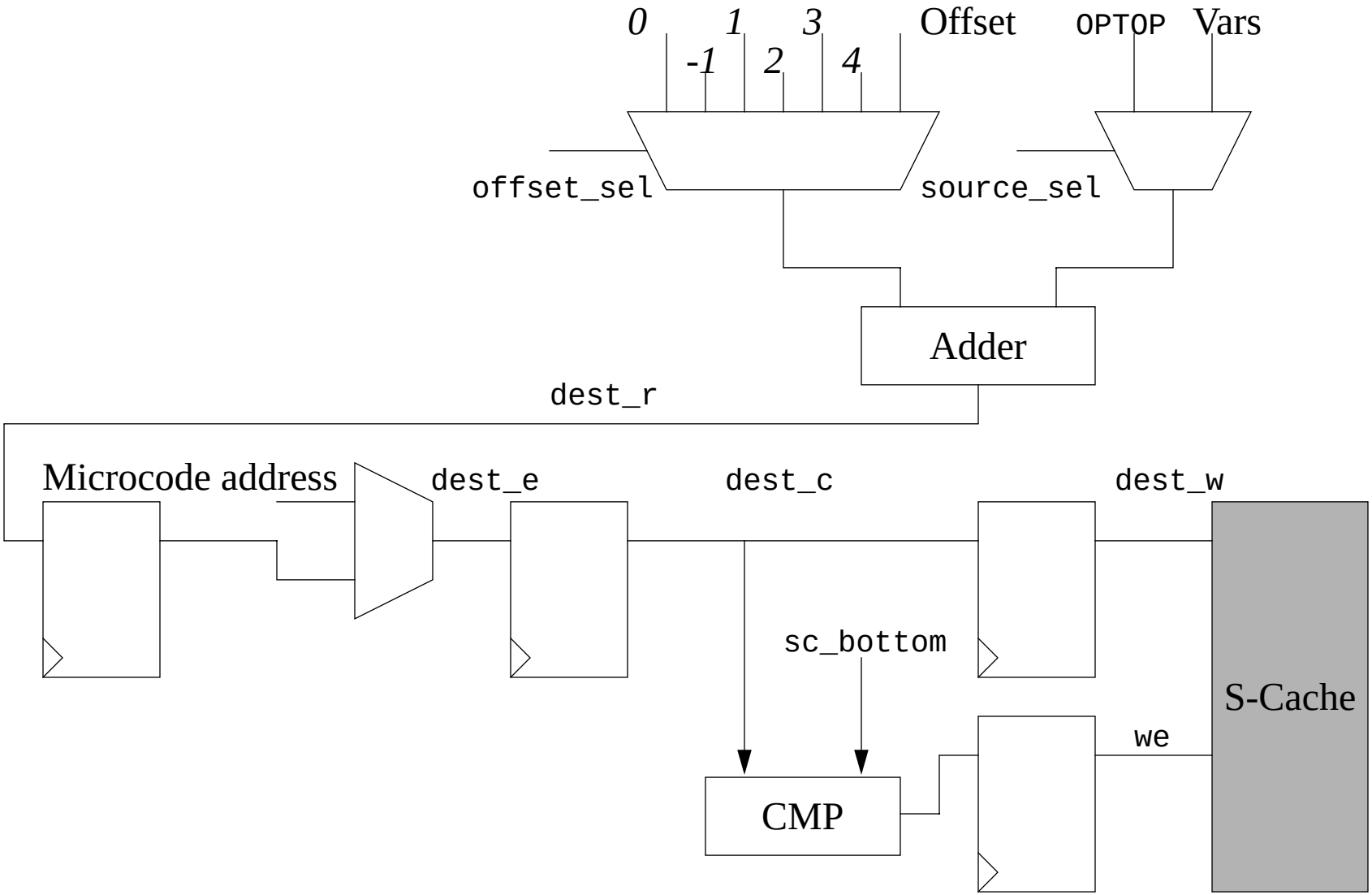
Register Access Logic



Bypass Logic



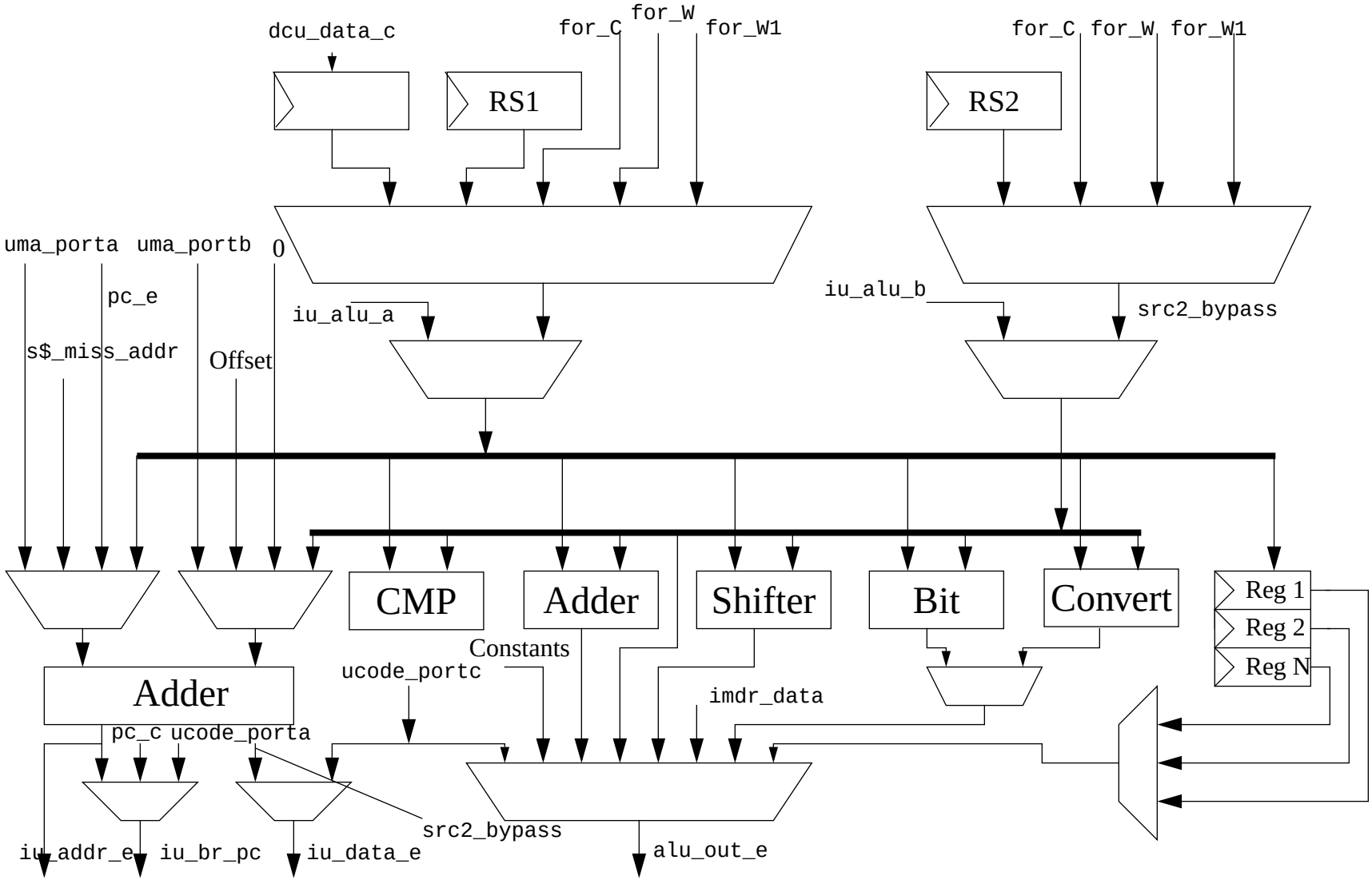
Destination Logic



IU Datapath

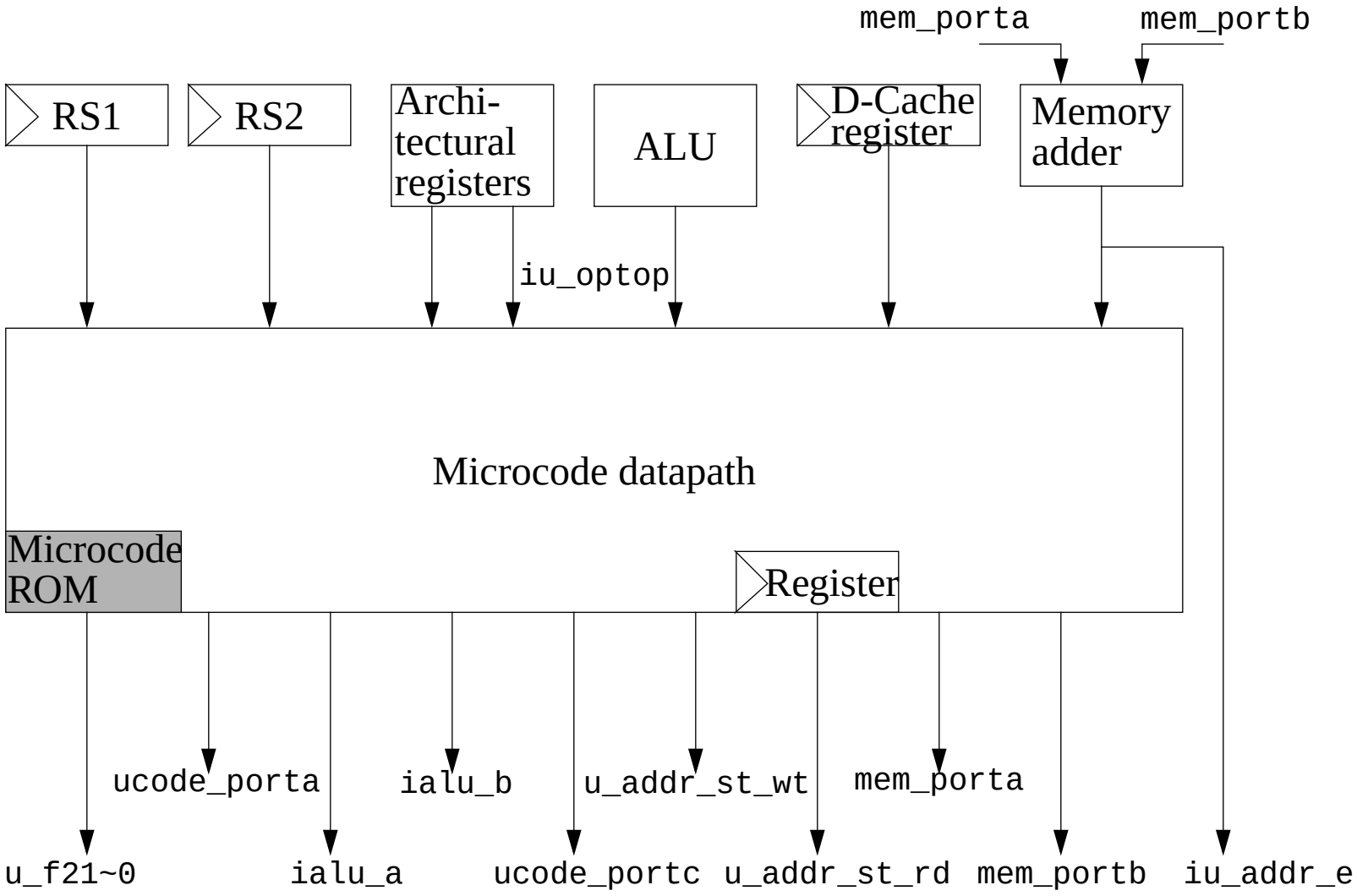
- **The IU datapath is a single-cycle execution engine used by simple instructions that take only one cycle as well as by the microcode for multicycle instructions and calculation of addresses.**
- **The IU datapath consists of the following:**
  - Main adder
  - Address adder
  - Comparator, shifter
  - Bit-wise operator
  - Integer converter
- **The IU also maintains most architectural registers inside the datapath, except for:**
  - `PC` and `OPTOP`, which it monitors in pipe control
  - `GLOBAL0`, `GLOBAL1`, `GLOBAL2`, and `GLOBAL3`, which it monitors in the R stage

IU Datapath

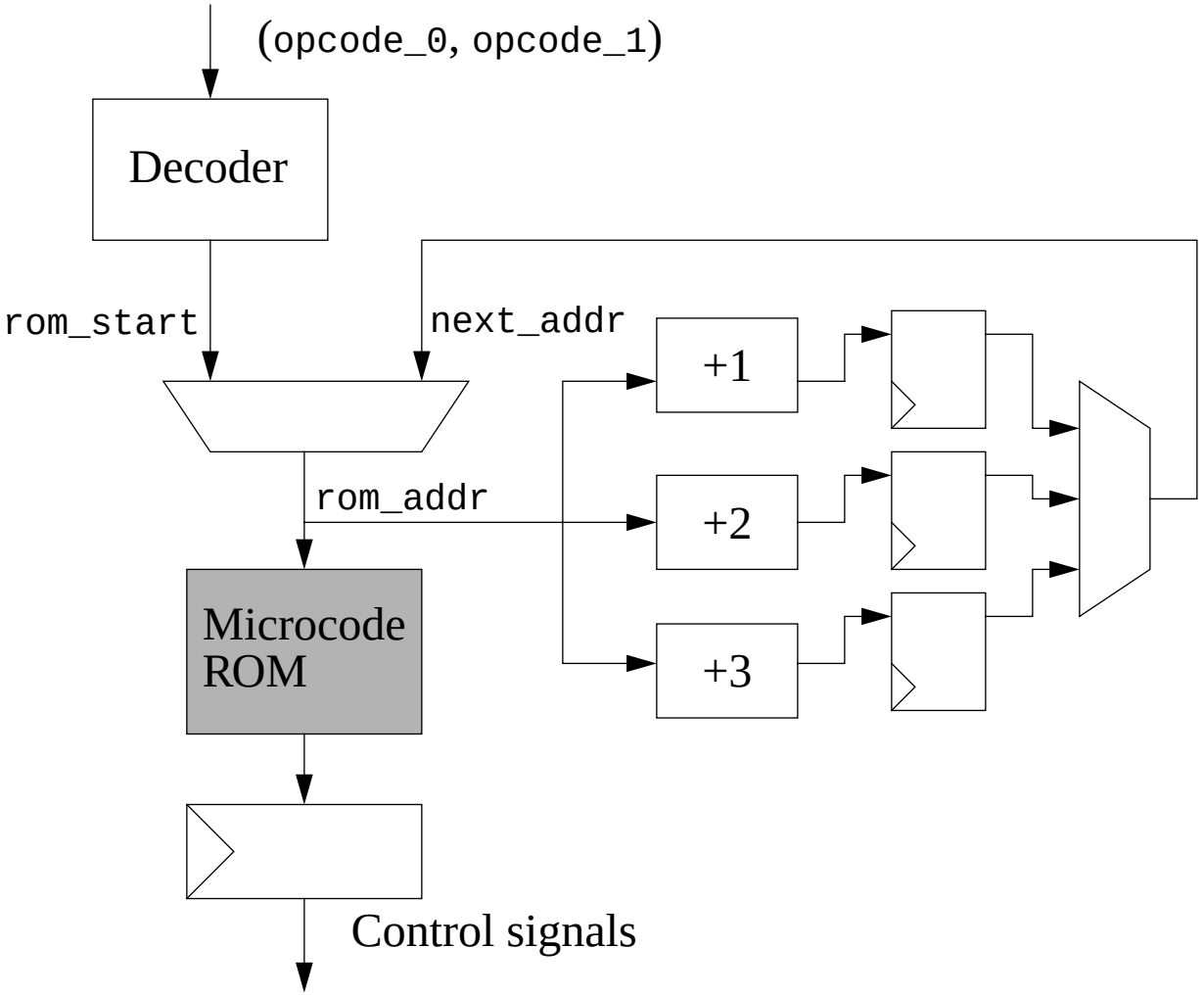


Note - This figure is a simplified version of the datapath. For more details, see *picoJava-II Microarchitecture Guide*.

Microcode


mem_porta
mem_portb
RS1
RS2
Archi-
tectural
registers
ALU
D-Cache
register
Memory
adder
iu_optop
Microcode datapath
Microcode
ROM
Register
ucode_porta
ialu_b
u_addr_st_wt
mem_porta
u_f21~0
ialu_a
ucode_portc
u_addr_st_rd
mem_portb
iu_addr_e

Microcode Control

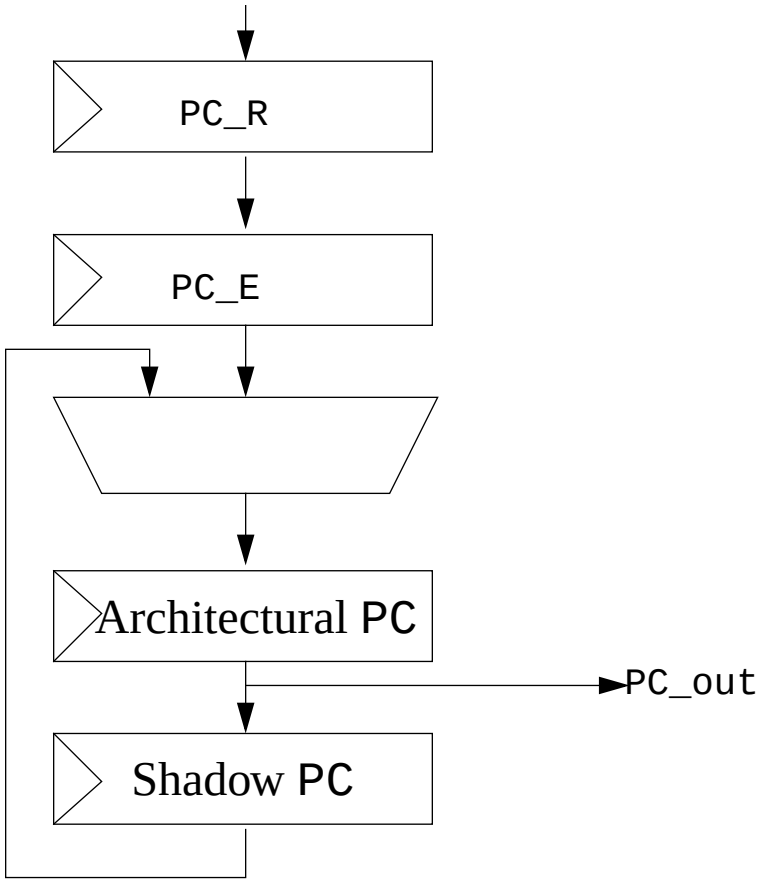


Microcode Control (Cont’d)

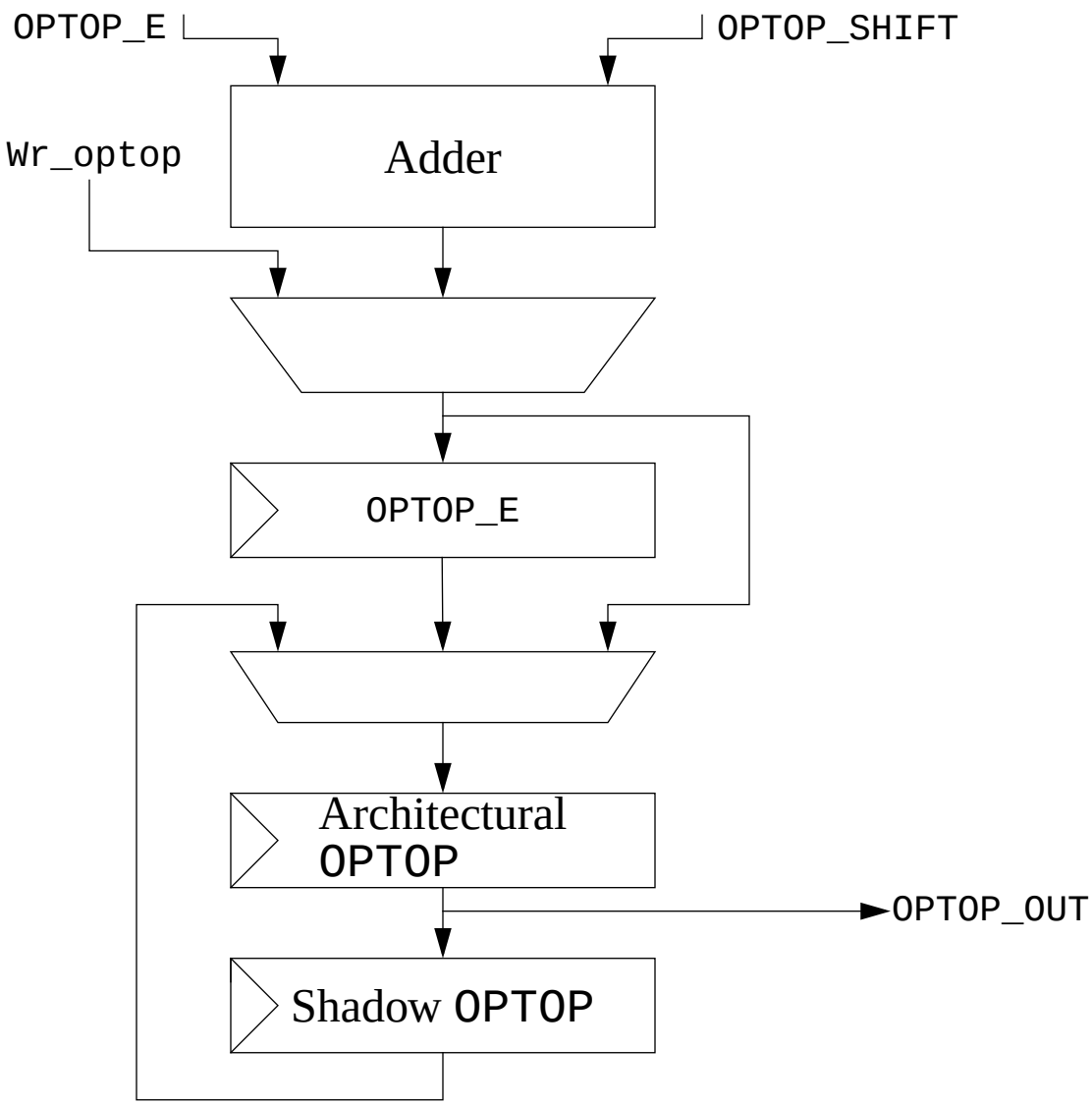
- **Microcode implements the multicycle operations.**
- **Microcode has its own datapath, which consists mainly of eight temporary registers and a ROM (284 x 70)**
- **Microcode shares two adders in the IU.**

Pipe Control Unit

- **The Pipe Control Unit keeps track of the PC and OPTOP registers.**



PC Datapath



OPTOP Datapath

Reissue Logic

- **If a folded group misses the S-Cache while accessing any of the two operands, pipe control resets `OPTOP` and `PC`.**

  It then reissues the folded group of instructions.

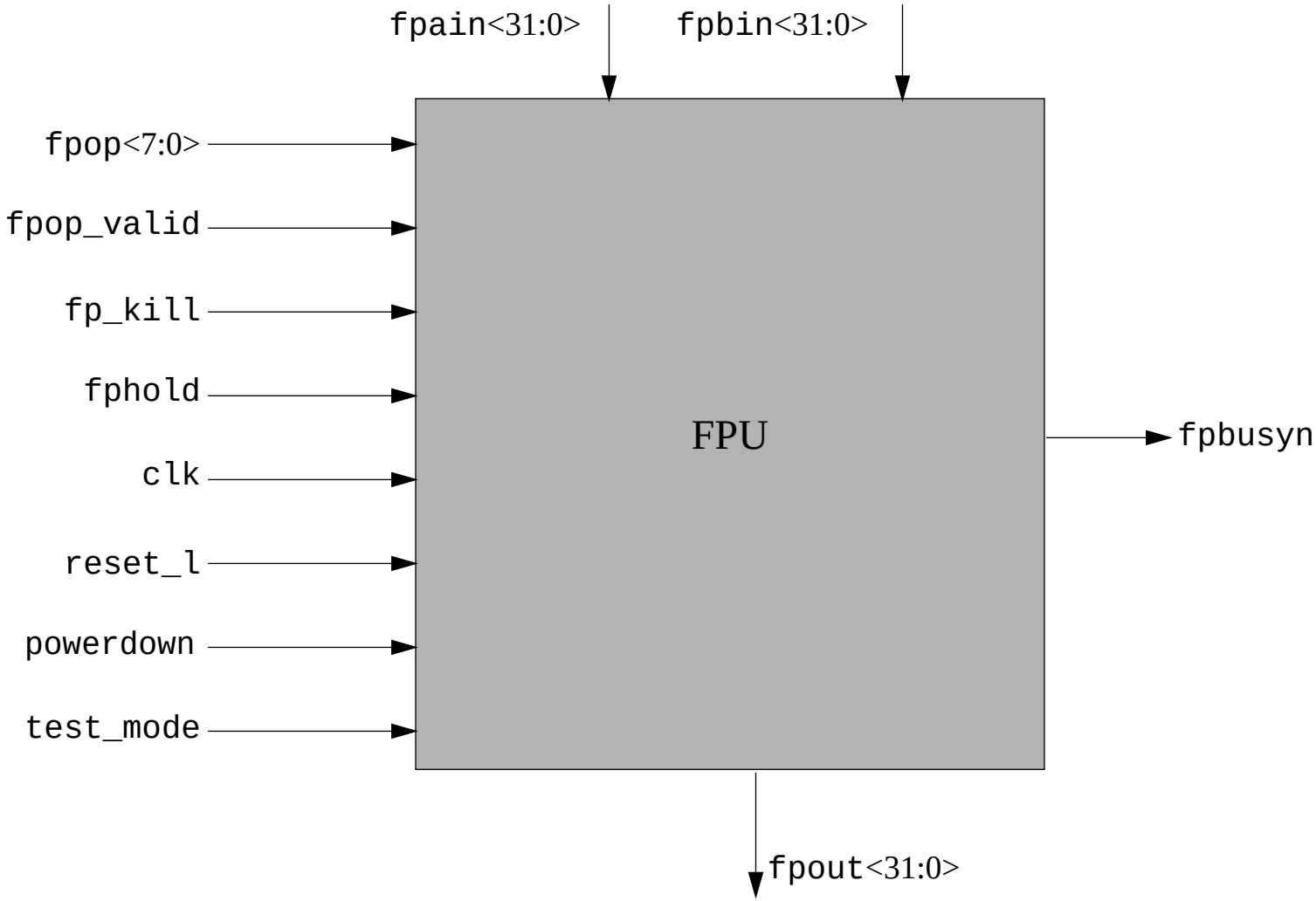
- **If a folded group traps, pipe control resets `OPTOP` and `PC`.**

  It then reissues the folded group of instructions with folding disabled for the next four instructions.

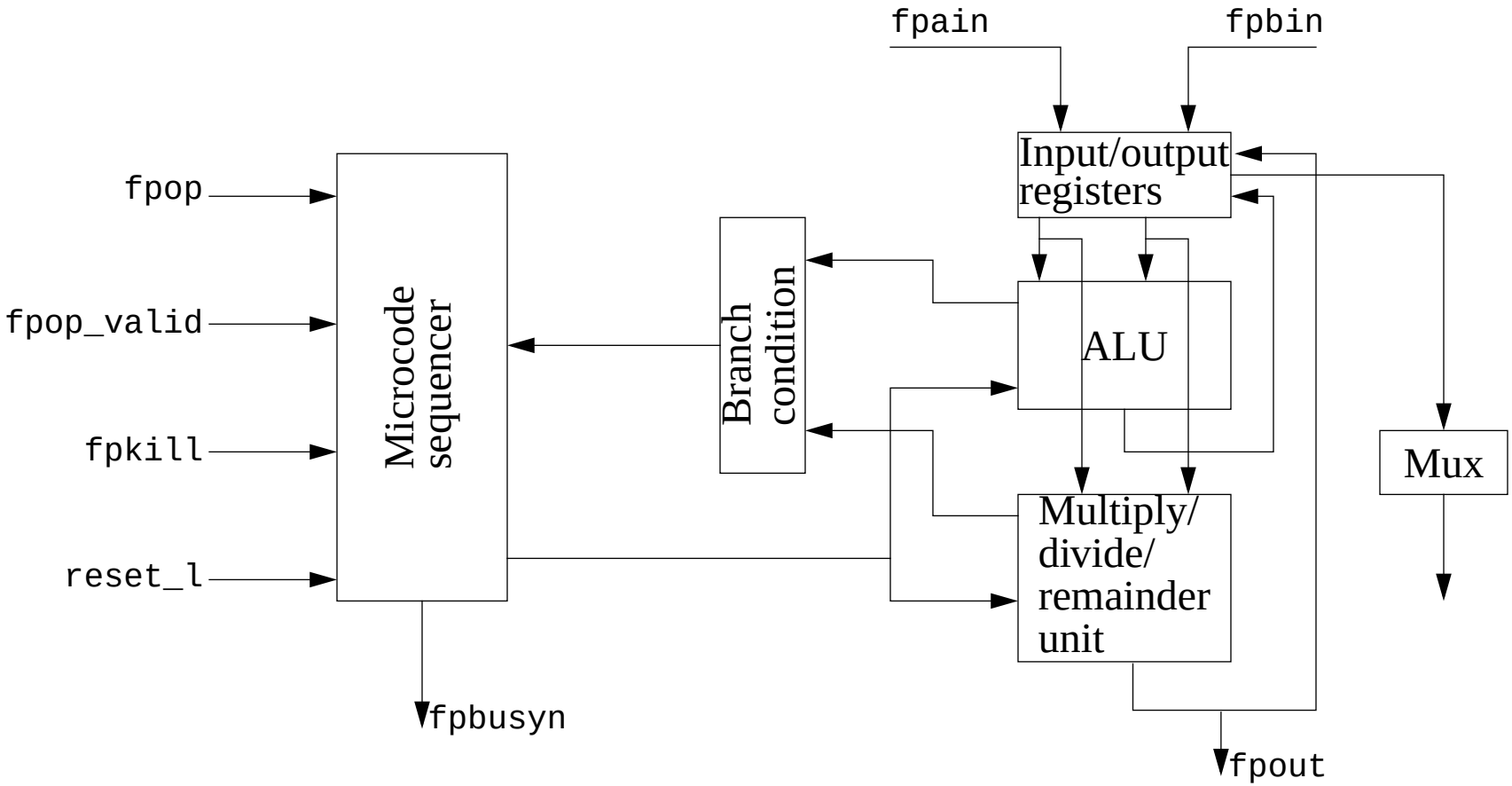
Stall Types

- **DCU hold** — Holds the DREC stages of the pipeline due to a load miss.
- **SMU hold** — Holds the DREC stages of the pipeline in an underflow or overflow.
- **Microcode busy** — Holds the DR stages of the pipeline when the microcode is active.
- **IMDR busy** — Holds the DR stages of the pipeline when the multiply/divide/remainder unit is busy.
- **FPU busy** — Holds the DR stages of the pipeline when the FPU is active.
- **S-Cache hold** — Holds the DR stages of the pipeline on a local variable load miss in the R stage.
- **ICU hold** — Holds the DR stages of the pipeline during diagnostic writes and reads to the I-Cache or I-Tag.
- **Multicycle hold** — Holds the D stage of the pipeline during the fetch of operands for long and double operations.
- **LDUSE hold** — Holds the pipeline for one cycle in a load use—a load in the E stage and a use in the R stage.

Floating Point Unit (FPU)



FPU Structure



FPU Structure (Cont'd)

- **Microcode sequencer** — Controls the microcode flow and microcode branches.
- **Input/Output Registers** — Controls input-output data transactions. Also provides the input data loading and output data unloading registers for intermediate result storage.
- **Floating point adder-ALU** — Includes the combinatorial logic that preforms floating-point adds, floating-point subtracts, and conversion operations.
- **Floating point multiply/divide/remainder unit** — Contains the hardware for performing multiply, divide, and remainder operations.

Stack Manager Unit (SMU)

The SMU does the following:

- **Moves data in and out of the stack into memory in an overflow or underflow in the stack cache**
- **Handles spills and fills of the stack cache by speculatively dribbling the data in and out of the stack cache from and to the data cache**
- **Generates a pipeline stall to stall the pipe in a stack cache overflow or underflow condition**
- **Keeps track of the requests to the data cache. A single request to the data cache consists of a 32-bit consecutive load or store request.**
- **Handles stack cache write misses of the IU**

Stack Cache

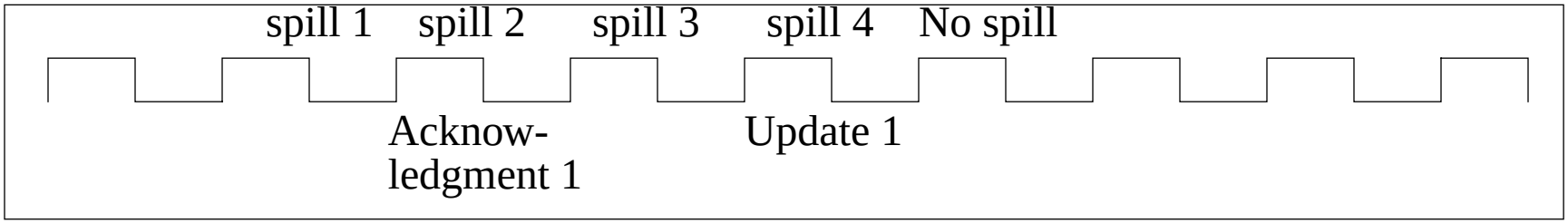
- **The stack cache is a 64-entry register file with three read ports and two write ports.**
- **The stack cache caches the top few entries of the operand stack.**
- **Two read ports and one write port fetch and store operands to and from the IU.**
- **Exclusive read and write ports perform background dribbling of data to and from the data cache.**
- **All memory addresses generated as offsets from `OPTOP`, `VARS`, or `FRAME` are stack accesses. All other memory accesses go directly to the data cache.**

Dribbling Operations

- **Spill**

(SC_BOTTOM – OPTOP) > high watermark

If the number of entries exceeds the high watermark, the SMU starts a spill transaction, sends a write request to the data cache along with the address and data, and receives an acknowledgment in return. It sends another request if a spill condition continues to exist.



- **Fill**

(SC_BOTTOM – OPTOP) < low watermark

If the number of entries is less than the low watermark, the SMU starts a fill transaction, sends a read request to the data cache along with the address, and receives an acknowledgment along with the requested data in return. It then writes the data into the stack cache and, if a fill condition continues to exist, sends another request.

Dribbling Operations (Cont'd)

- **Overflow**

$$(\texttt{SC\_BOTTOM} - \texttt{OPTOP}) > 60$$

If the number of entries on the stack is greater than 60, a stack overflow occurs. Four empty stack locations handle interrupts.

The SMU then stalls the pipeline and activates a state machine, which sends a series of write requests to the data cache.

Once the dirty entries have been written into the data cache, the stack cache verifies that the six top most entries of the stack (based on the new `OPTOP` location) are present in the stack cache, reading them from either the data cache or memory.

When there are six entries in the stack cache, SMU deasserts the stall. Execution of instructions continues.

- **Underflow**

$$(\texttt{SC\_BOTTOM} - \texttt{OPTOP}) < 6$$

If the number of entries on the stack cache is less than six, a stack underflow occurs. All stack cache accesses generated as offsets of `OPTOP` are considered stack hits.

Dribbling Operations (Cont’d)

The SMU then stalls the pipeline and activates a state machine, which sets `SC_BOTTOM` to the new `OPTOP` value and sends read requests until there are six entries in the stack cache.

Once the SMU has written the data onto the stack cache, it deasserts the stall. Any entries beyond `OPTOP` are not used and, therefore, not saved.

The SMU can activate an underflow only in response to changes in `OPTOP` caused by the return instructions.

- **Stack cache write misses**

```
dest_address > SC_BOTTOM
```

When the IU has a stack cache miss on a write, it issues the SMU the store address and the data. The SMU gives this request higher priority over spills and fills.

The SMU holds the pipe until it has issued the store to the data cache, then clears its pipe, issues the miss address, and stores the miss data to the data cache.

Data Cache Unit (DCU)

- **The DCU does the following:**
  - Arbitrates data cache requests from the SMU and the pipeline
  - Generates address, data, and control signals for the data and tag RAMs
  - Reorders the data RAM output to provide the correct data in a cache hit
  - Provides datapath and control logic for processing of noncacheable requests
  - Provides the datapath and datapath control functions for cache misses
- **The data cache is a two-way set associative, write-back, write-allocate, 16-byte line cache. The cache size is configurable to 0, 1, 2, 4, 8 and 16 Kbytes.**
- **Each line has a cache tag store entry associated with it. On a cache miss, the DCU write 16 bytes of data into the cache from main memory.**
- **The arbiter arbitrates data cache requests from the pipeline and the SMU. Usually, the pipeline has a higher priority than the SMU. However, in cases where the SMU holds the pipe, it has a priority over the IU pipe.**

  The arbiter selects address and data inputs for the tag and data RAMs.

Data Cache Unit (DCU) (Cont'd)

- **Address control updates the following:**
  - The address fields of the tags (while writing to the caches)
  - The LRU bit during cache access, invalidating the entry in case of flushes
  - The dirty bit while writing to the caches
- **Aligner control provides the following:**
  - Write enable signals to the data RAM
  - Control signals (to align data from the data RAM during cache hits)
- **Miss control does the following:**
  - Interfaces with the write back control and read-buffer control
  - Handles cache misses
  - Generates a stall signal to the pipeline as well as bus requests for cache line fills
  - Provides handshaking signals for the above transactions
- **Writeback control determines which line to replace, depending on the LRU bit.**

  If the dirty bit of the line is set, writeback control moves that line into write buffer. Once the cache fill transaction started by miss control is complete, writeback control starts a writeback cycle.

Data Cache Unit (DCU) (Cont’d)

- **The data cache datapath does the following:**
  - Provides the appropriate address and data to the RAMs
  - Post-processes data before sending them to the pipeline or the register file
  - Sends the appropriate address and data on the memory buses for cache fill and write-back transactions

  The datapath also contains a single 16-byte buffer to accommodate a cache line that is being replaced.

Data Cache Unit (DCU) Transactions

- **Cache reads**

  In the C stage of the pipe, the DCU accesses the tags and data RAMs with the read address, then compares the tags. In case of a cache hit, the load's data are available at the end of the C stage after going through the aligner muxes. In case of a cache miss, the DCU makes a request to the BIU.

- **Cache writes**

  In a cache hit in the C stage, the DCU writes data into the cache in the C + 1 stage. In a cache miss, since the data cache is a write-allocate writeback cache, the DCU fetches a line from the memory and writes it into the D-Cache.

  Since the SMU writes sequential data, if the write to the end of a cache line misses the D-Cache, the DCU writes the missed data directly into that cache instead of reading from memory. This protocol saves memory cycles and context switch time as well as improves the interrupt latency.

  DCU does not stall the pipeline on a cache write miss. However, it cannot accept additional requests until the store transaction is complete. The IU or SMU can dispatch back-to-back stores for the data cache.

  There is a bubble between a store and an immediate load.

Data Cache Unit (DCU) Transactions (Cont'd)

- **Cache Fills**

  On a cache miss, the DCU sends a cache fill request to the memory controller at the end of the C stage.

  A cache fill transaction starts in a cache miss where there are no outstanding requests.

  The DCU waits until it receives a `pj_ack` signal from the memory controller, then writes the data from the memory bus onto the cache.

- **Writebacks**

  A writeback transaction takes place in a cache miss, where there is a dirty line to be replaced.

  In cacheable loads and stores, writeback transactions start after a cache line fill cycle is complete.

  Writebacks take place in the back ground. Thus, the pipeline stall is deactivated once the cache fill transaction is complete.

Data Cache Unit (DCU) Transactions (Cont’d)

In case another miss occurs while a writeback is in progress, the cache fill transaction waits for it to finish before starting its process.

If there is a request for the line being written back during the writeback cycle, the DCU does not bypass that data. Instead, it generates a cache miss.

- **Noncacheable (NC) loads**

  Noncacheable loads force a cache miss; the DCU makes an NC request to the bus. Once the data are available, the DCU bypasses them to the pipeline.

- **Noncacheable (NC) stores**

  Noncacheable stores also force a cache miss; the DCU makes an NC request to the bus. The DCU then writes the data into the writeback buffer and does not stall the pipeline. This transaction is very similar to a writeback once the store data are written into the write buffer.

  If the write buffer contains a cache line, a noncacheable store occurs after the cached store is retired, thus maintaining strong write ordering.

Data Cache Unit (DCU) Transactions (Cont'd)

- **Cache compare flushing**

  In a cache compare flushing operation, the DCU accesses the corresponding line tags and compares them to determine whether there is a cache hit. If so and the dirty bit is not set, the DCU turns off the tag's valid bit. If the dirty bit is set, the DCU starts a writeback transaction and updates the `LRU` bit.

- **Cache indexed flushing**

  Cache indexed flushing is similar to cache compare flushing, except that the DCU does not compare the line tags. If the cache line is dirty, the DCU writes it back to memory. The DCU resets the valid bit after moving the dirty line into memory.

- **Cache invalidate flushing**

  In a cache invalidate flushing, the DCU accesses the corresponding line tags and compares them to determine if there is a cache hit. If so, the DCU invalidates the line and does not start writeback transactions. If there is a cache miss, no state change occurs.

- **Zeroing of cache lines**

  The `zero_line` instruction takes five cycles to complete—one to determine hit or miss and four to write. The IU assumes this to be a one-cycle operation and continues

Data Cache Unit (DCU) Transactions (Cont'd)

the execution flow unless there is an instruction following `zero_line` that uses the data cache, in which case the DCU stalls that instruction until `zero_line` completes.

The DCU checks the tags to determine if the cache line is a valid dirty line. If so, the DCU initiates a writeback transaction and zeroes out the cache line. It also updates the `LRU` bit and sets the dirty bit. If not, the DCU fills the cache line with zeroes.

- **Nonallocated writes**

  Nonallocated writes are processed in a manner similar to cacheable stores on a cache hit. On a cache miss, the DCU converts nonallocated writes into noncacheable stores and sends them to memory.

  The DCU uses nonallocating writes for the SMU request to improve dribbler store performance.

- **Diagnostic reads and writes**

  The DCU can use privileged instructions to perform diagnostic reads and writes to both the data cache RAM and the tags.

Data Cache Unit (DCU) Transactions (Cont’d)

D-Cache or tag access address format

D-Cache data format

D-Cache data format

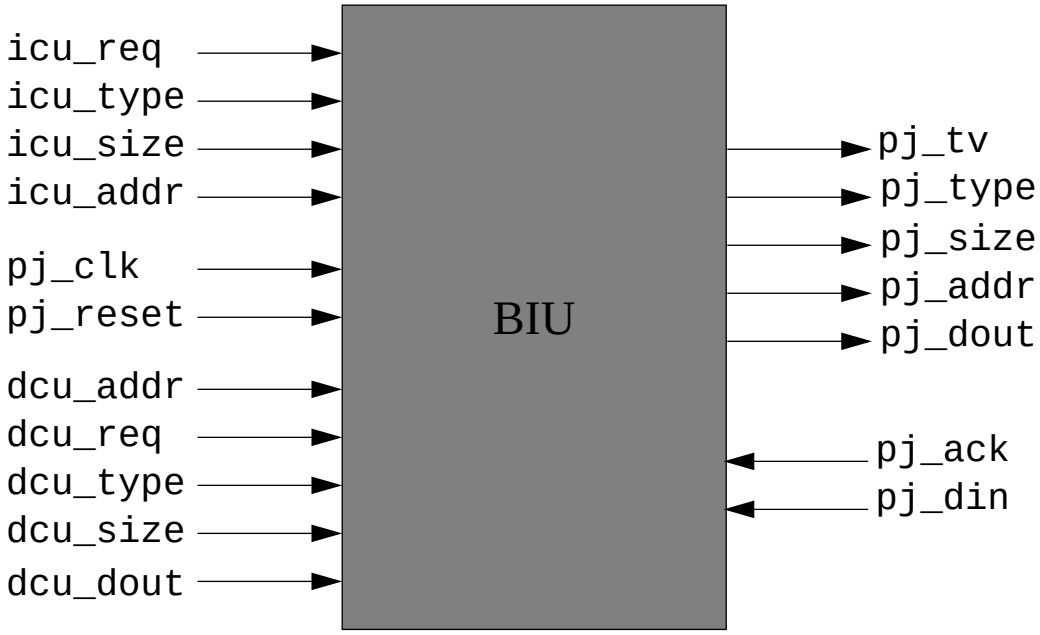
Bus Interface Unit (BIU)

The BIU contains arbitration logic for picoJava-II internal requests and performs the following tasks:

- **Generates read and write requests to memory**
- **Provides acknowledgments to the I-Cache and data cache controllers so that they can sync data**
- **Handle errors that are generated on the memory bus**
- **Arbitrates internal requests from the instruction cache and data cache units and generates external requests on the bus**

  Data cache requests have a higher priority than instruction cache requests.

Bus Interface Unit (BIU) (Cont'd)



BIU Arbiter