DRAFT: Do not distribute

MIT Concurrent VLSI Architecture Memo ??

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

The MAP Instruction Fetch Unit (IFU)

William Dally

August 4, 1995

Abstract

This document describes the logic design and Verilog realization of the instruction fetch unit (IFU) of the multi-ALU processor (MAP) chip.

Contents

1 Overview **2**

2 Data Paths **2**

3 External Interfaces **3**

4 Operation **6**

4.1 Normal Operation 6

4.2 I-Cache Miss Request 8

4.3 I-Cache Miss Response 9

4.4 Miss Conflict 10

4.5 Branch 10

4.6 Thread Restart 12

5 Detailed Description **13**

5.1 Instruction-Pointer Datapath 13

5.2 Instruction-Pointer Control Logic 17

5.3 Instruction Cache and Control 18

5.4 Miss Handler 19

5.5 Operation-Buffer Datapath 20

5.6 Operation Buffer Control 23

A Overview Diagram **24**

B IFU Datapath Timing Specifications **25**

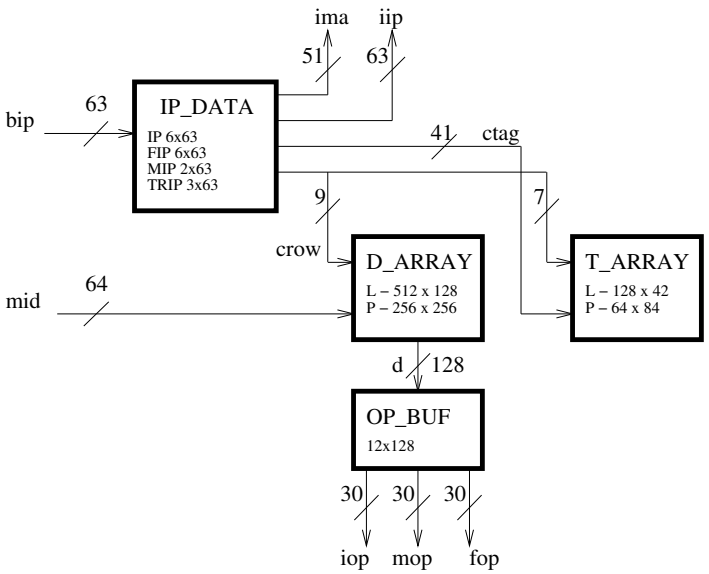

Figure 1: *IFU Datapath Block Diagram*

1 Overview

The instruction fetch unit (IFU) fetches instructions from memory, caches them in a local (per cluster) 8KByte I-cache, and formats the variable length instructions into a standard fixed-width format for presentation to the pipeline. To carry out this function, the IFU maintains instruction pointers and operation buffers for each thread.

2 Data Paths

Figure 1 shows the datapaths of the IFU.

All addresses enter the ifu on the `bip` bus. This 63-bit bus holds tagged half-word addresses (low two bits omitted). Branch addresses are passed on `bip` as are addresses to fork and restart threads.

Addresses passed on `bip` are stored in the registers of the instruction-pointer datapath `IP_DATA`. This datapath contains 17 63-bit registers including:

- Six fetch instruction pointers (FIPs) that point to the next cache line to be fetched for each thread.
- Six instruction pointers (IPs) that point to the next operation to be issued to the register read stage (RR).

- Two miss instruction pointers (MIPs) that hold the FIP during a miss. This is required because a branch during a miss overwrites the missing FIP which is needed to load the IP after the next instruction in that thread issues.
- Three thread restart instruction pointers (TRIPs) that hold the IPs to be loaded into the front end of the pipeline when a thread is restarted.

The IP_DATA block outputs three addresses:

- IFU to M-Switch Address `ima` this 51-bit word address (low three bits omitted) is used to fetch instruction words from the memory system during an I-cache miss.
- Issue Instruction Pointer `iip` this 63-bit address is the pointer to the instruction being issued to the RR stage. The low three bits of this address select the next 32-bit operation from the eight operations in the selected thread's operation buffer.
- Cache instruction pointer `cip` is a 50-bit double-word (low four bits omitted) instruction that is used to address the cache. It is broken into two components. A 9-bit hash of the bits is used as a cache row address `crow` while the high 41 bits are used as the cache tag `ctag`. The high 7-bits of `crow` are used to address the tag array while `ctag` is compared against the tag stored at that address.

The `crow` signal addresses the two cache arrays: `D_ARRAY` and `T_ARRAY`. The data array is logicaly a 512 by 128 SRAM that is word writable. With 2:1 multiplexing it will physically be a 256 row by 256 column SRAM. On cache misses it is written by the memory to IFU data signal, `mid`, one word at a time.

The tag array is addressed by the top seven bits of `crow` since there is one tag for each four double words (eight word line) of the data array. It is logically a 128 by 42-bit SRAM with a group clear on bit 41, the *valid* bit. With 2:1 multiplexing it will physically be a 64 row by 84 column SRAM with group clear on 2 columns. The tag array includes a comparator that signals a match if `ctag` equals the entry stored at the addressed row. On cache misses `ctag` is written into the addressed row when the last word of the row has been written to the data array.

The 128-bit data output of the D_ARRAY, `d`, is fed to the operation buffer. This block contains 12 128-bit instruction buffers, two for each thread. The pair of buffers of the selected thread are driven onto a 256-bit (eight operation) bus and the next three operations are selected by an operation shifter. An operation swizzler then selects the appropriate fields to be driven to the integer operation, `iop`, memory operation, `mop`, and floating operation, fop, outputs.

3 External Interfaces

The IF unit provides five main interfaces: instruction output, thread select, branch control, miss request, and miss response. The interface signals are listed in the following tables.

Each signal name is listed along with its timing, direction, and a brief description. Timing is denoted as one of:

B valid on both phases. The signal is valid early in the cycle and held until the rising edge of the clock.

1 stable on phase 1. The signal is stable by the rising edge of the clock and is held until the falling edge of the clock.

2 stable on phase 2. The signal is stable by the falling edge of the clock and is held until the rising edge of the clock.

v2 valid on phase 2. The signal is valid by the rising edge of the clock.

Thread Select:

Name	Timing	Dir	Description
`avail[5:0]`	B	O	The corresponding thread may be available for issue. A bit of this signal is asserted if there is a chance that an instruction can be issued from a thread. If this thread is selected and the IFU later discovers that the instruction cannot be issued, the `ikill` signal is asserted to abort the issue. This can occur, for example, on an I-cache miss or when an instruction unexpectedly wraps around the end of a 128-bit instruction buffer. The `avail` signal is valid early in the cycle and held constant until the rising edge of the clock (B).
`brok[5:0]`	B	O	The corresponding thread is allowed to branch. A bit of this signal is asserted if the corresponding thread can safely branch. For a thread to branch, the corresponding `mip` register must be empty to catch the last delay slot `fip` if it misses and there must be sufficient information in the instruction buffer so that the last delay slot instruction can be fetched in a single I-cache reference even if the instruction wraps. The `brok` signal is valid early in the cycle and held till `clk` rises.
`halt[5:0]`	B	I	Thread halt from the sync stage (SZ). A bit of this signal when low enables the corresponding thread to fetch instructions.
`ts[5:0]`	1	I	Thread select from SZ. Bit i of this signal is asserted to select an available (`avail[i]`=1) thread for issue. This signal is required to be valid about 2ns before the end of the clock cycle.

Instruction Output:

Name	Timing	Dir	Description
`iop[29:0]`	1	O	Integer operation of the current instruction. This operation is only valid if (1) a bit of `ts` is asserted, (2) `ikill` is not asserted, (3) `iv` is asserted, and (4) `illegal` is not asserted.
`mop[29:0]`	1	O	Memory operation.
`fop[29:0]`	1	O	Floating operation.
`iv`	1	O	Integer operation valid.
`mv`	1	O	Memory operation valid.
`fv`	1	O	Floating operation valid.
`illegal`	1	O	The current instruction is illegal.
`iip[62:0]`	1	O	The instruction pointer for the current instruction.
`ikill`	v2	O	The current instruction is not valid, kill the issue.

Branch Control:

Name	Timing	Dir	Description
`bs[5:0]`	B	I	Branch select. Bit i of this signal when asserted causes thread i to branch to the specified IP **after one more operation is issued**. Branch select must be valid early in the cycle and held until the rising edge of the clock. If a thread is *empty* a branch causes a fork setting both the ip and fip.
`trld`	B	I	Thread restart load. When asserted in conjunction with `bs` this signal initiates a thread restart operation and loads the first thread restart IP (TRIP) on the `bip` bus. It must be asserted three more times to load the remaining three TRIPs. The thread is then primed for a restart.
`trdone`	B	O	Thread restart done. When asserted this signal indicates than any pending thread restart operation is complete and the IFU is ready to handle a new thread restart operation.
`bip[62:0]`	v2	I	Branch instruction pointer. Used to pass addresses into the IFU for branches, forks, and thread restarts.

Miss Request:

Name	Timing	Dir	Description
imreq	B	O	IFU to memory request. When asserted, the IFU is requesting a memory read with address and token given by the ima and imtok lines. A request will be repeated until it is acknowledged.
imack	v2	I	IFU to memory acknowledge. The C-switch interface asserts this signal to indicate that a request on the imreq line has been accepted.
ima[50:0]	B	O	IFU to memory address. The 51-bit word address for memory reads requested on imreq.
imtok[3:0]	B	O	IFU to memory token. This 4-bit token identifies the memory request and must be returned by the memory system with the requested data.
.			

Miss Response:

Name	Timing	Dir	Description
mireq	B	I	Memory to IFU request. When asserted, the memory system is returning data on the mid lines and associated token on the mitok line. The IFU must accept the request. This signal must be asserted early in the cycle to provide control to perform the cache write of the returned data.
mid[63:0]	2	I	Memory to IFU data. Data on these lines is written to the I-cache when mireq is asserted. Note that the pointer bit and sync bit from the C-bus are not used here.
mitok[3:0]	B	I	Memory to IFU token. The token provided with the address during the memory request is returned on these lines to identify the associated data.

4 Operation

4.1 Normal Operation

Figure 2 illustrates the timing of the instruction fetch unit during a normal fetch and one terminated by a cache miss.

At the end of each cycle an arbitration is performed to select which thread, if any, will perform an instruction cache access on the next cycle. The result of this arbitration is latched on the rising edge of the clock and output radially on the cs lines one of which is depicted in the figure. The cs lines select which fetch instruction pointers fip is to be used as the cache instruction pointer cip from which the crow and ctag signals are derived. The fip signal is updated on clk high, so after clk rises the selected address must propagate

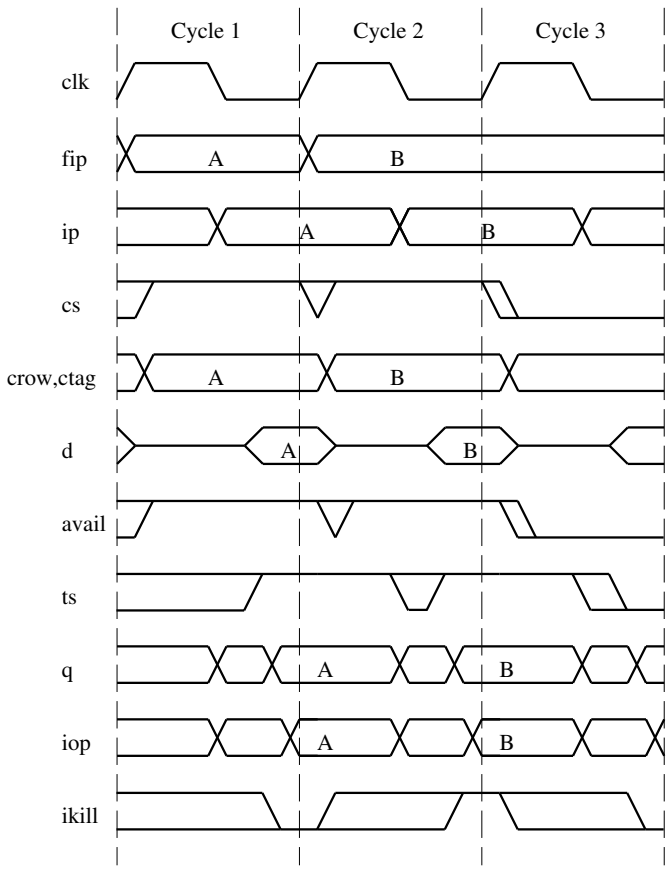

Figure 2: *Normal Instruction Fetch Sequence*

through the latch, the cs mux, a hash, and across some metal lines before reaching the cache address pins.

The cip is used access the instruction cache resulting in the cache data output d being valid near the end of the cycle. The d output is latched into the appropriate operation buffer (selected by cs and the low bit of cip) which is transparent on clock low. The match line from the tag array also becomes valid near the end of the cycle. During Cycle 1 the match line is high at the end of the cycle so the access is not killed. Cycle 2 depicts a cache miss in which the present instruction wraps into the missed cache row causing the access to be killed.

The thread available avail signals (one per thread) signal to the synchronization stage which threads are likely to have an instruction ready to issue to the RR stage this cycle. If an instruction unexpectedly wraps around the end of an instruction buffer or if a cache access misses, a thread which was marked available may be unable to issue an instruction. If this thread is selected by ts the cycle will be killed.

The synchronization stage arbitrates among the *available* threads and generates the one-hot thread select or ts signals to select the thread, if any, to issue an instruction this cycle. The ts signals must be asserted early enough in the second phase to allow signals to be derived from them before the rising edge of the clock. The ts signals must be held steady through phase 1.

The ts signals select one thread's operation buffers to be driven onto the 256-bit wide q bus and select one of the instruction pointers ips to be driven onto the iip signal. The iip signal has timing identical to q. The low bits of iip select which half-word of the instruction buffer contains the next operation to be issued from the q bus. The instruction swizzler uses these bits to extract the next three operations on the o0, o1, and o2 buses. The instruction length len is derived from the high two bits (the *pack bits*) of o0 and o1 along with the format of the instruction. This information is used by the instruction shifter to multiplex the proper operations on to the iop, fop, and mop outputs and generate the iv, mv, and fv signals. The timing of all of these signals is identical to that shown for iop.

4.2 I-Cache Miss Request

Figure 3 illustrates the timing of the memory requests initiated in response to a cache miss.

A miss is detected when the cache is being referenced (one of the six cs lines asserted) and match is low at the end of the cycle. The low match line prevents the fetch unit state (in particular the count of available operations) from being updated. If the miss prevents the IFU from issuing an instruction from the selected thread, ikill is asserted to abort the fetch.

There are two miss handlers. Threads 0-4 are handled by mh0 while thread 5 is handled by mh1. This allows the system thread (thread 5) to handle cache misses while the user and exception threads are locked up on, for example a TLB miss. If a thread's miss handler is

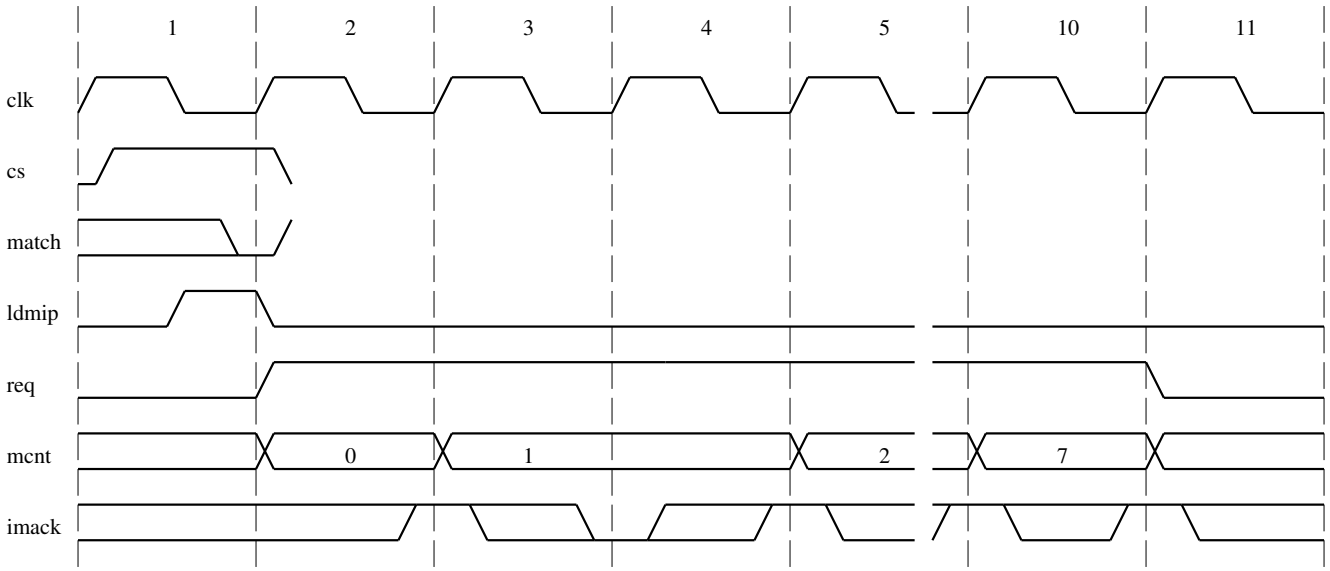

Figure 3: *I-Cache Miss Request Sequence*

busy handling a different thread when a thread takes a miss the miss is simply ignored and the access will be retried when the miss handler completes. Retrying the access prevents a line from being loaded from memory twice when two threads miss on it.

If the miss handler is idle when a miss occurs it will have captured the cache address `cip` since the `ldmip` signal is asserted on clock low whenever the handler is idle as shown in Cycle 1 of Figure 3.

On the cycle after the miss, the miss handler starts to make memory requests. Starting with the requested word first, the miss handler places addresses on the `ima` lines and asserts `imreq` (The addresses are derived from the count `mcnt` and the request is derived from the internal `req` signal both shown in Figure 3). If the M-switch accepts the operation `imack` is asserted and the count increments on the next cycle (as in Cycle 2). If the operation is not accepted, `imack` is low, and the count does not change at the end of the cycle (as in Cycle 3). After requests for all eight words of the line have been accepted, the `req` line drops and the `rec` line is asserted to indicate that the miss handler is in the receive state.

4.3 I-Cache Miss Response

The IFUs handling of words returning from memory during a cache miss is shown in Figure 4.

When the C-switch has a word for the IFU it asserts the `mireq` signal and places the data and token on `mid` and `mitok` respectively. The words may return from the memory system in a different order than they were requested. When `mireq` is asserted, the IFU forms the address by appending the low bits of `mitok` to the appropriate `mip` register and writes the

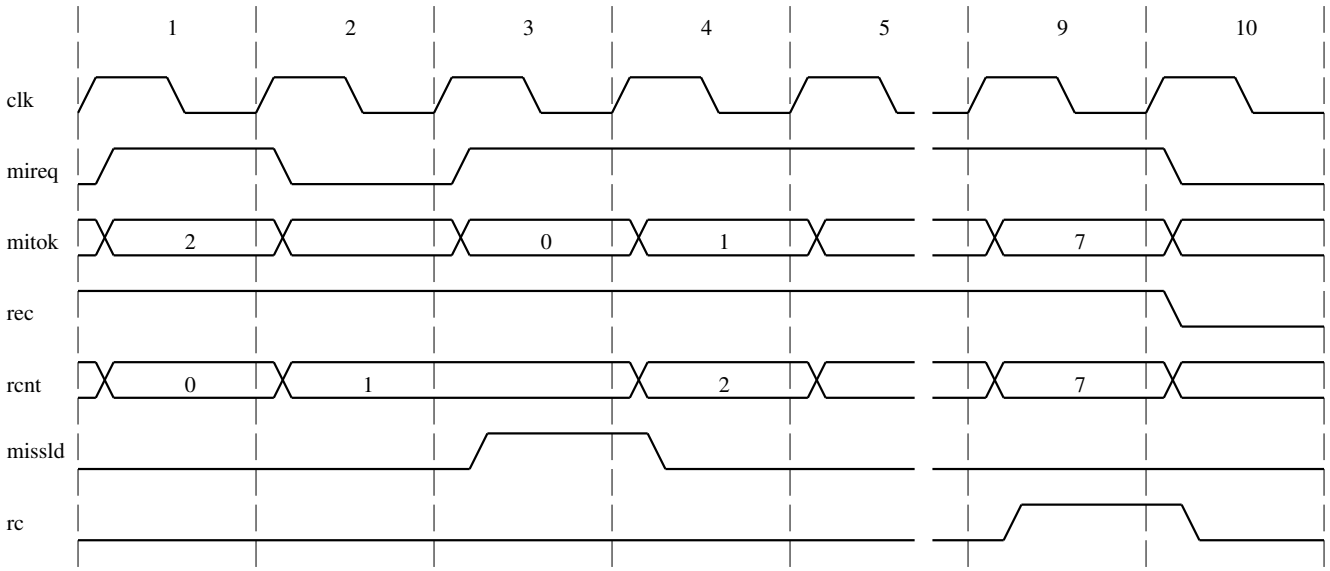

Figure 4: *I-Cache Miss Response Sequence*

data on `mid` to the appropriate cache line. When the second word of the requested row (not necessarily the one with the higher address as they may come back in either order) arrives, signal `missld` is asserted (as in Cycle 3) to load the data into the operation buffer. The `rcnt` signal keeps count of the number of words received so far. When the eighth word is received, signal `rc` is asserted to clear the miss register and write the tag into the cache tag array. On the next cycle signal `rec` goes low to indicate that the miss handler is idle.

4.4 Miss Conflict

A miss conflict occurs when a miss handler starts processing a miss to a cache line that the other miss handler is in the process of filling. If miss handler 0 is already filling a cache line and miss handler 1 takes a miss on an address that maps to this line, the `conflict` signal is asserted to disable further writes from miss handler 0.

If miss handler 1 is already filling a cache line and miss handler 0 takes a miss to an address that maps to this line, miss handler 0 is inhibited from processing the miss by asserting the signal `bcon` (backwards conflict).

4.5 Branch

Figure 5 shows the timing of a series of three branch operations.

To signal a branch on a particular thread, the corresponding `bs` signal must be asserted **at the beginning of the cycle** and the branch pointer, `bip`, must be asserted a setup time

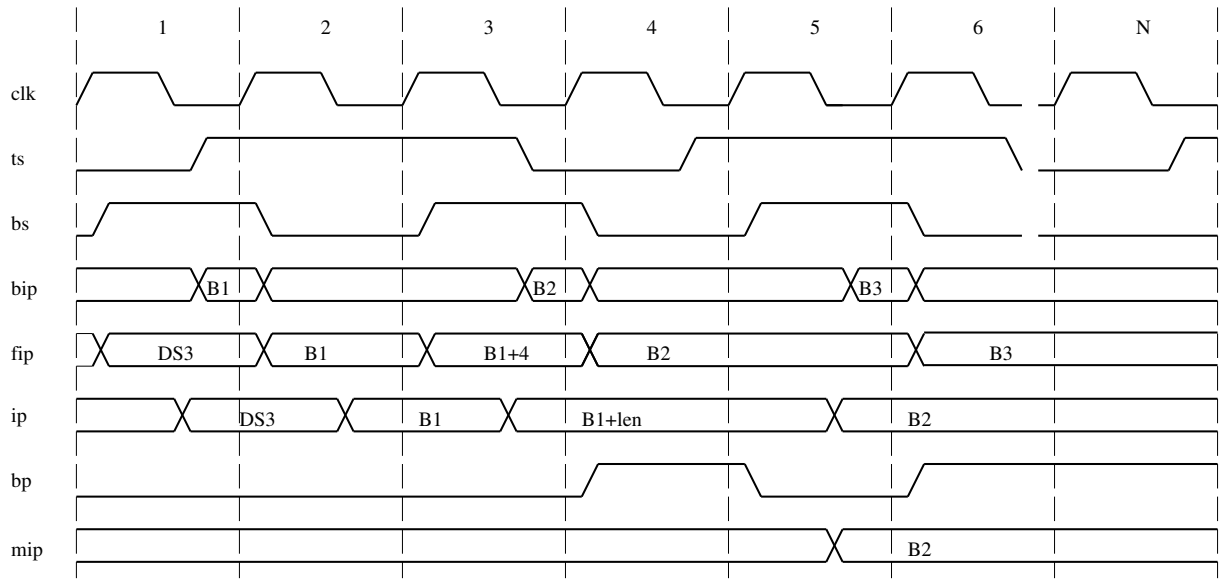

Figure 5: *Branch, Pending Branch, and Killed Branch Sequences*

before the end of the cycle. The `bs` signal must be asserted early to participate in cache arbitration. A branching thread takes priority in using the cache so the last delay slot will be fetched before the `fip` is overwritten.

A branch can only be performed if the branch-OK `brok` signal for the corresponding thread is asserted. When this signal is asserted the miss handler is idle so there is a `mip` register available to hold the `fip` in the event of a cache miss and there are adequate words in the operation buffer so that a single cache access will guarantee enough words to supply a three operation instruction.

In all three branches asserting `bs` causes the corresponding fetch instruction pointer `fip` to be loaded with the `bip` at the beginning of the next cycle. This allows the `bip` to be used to address the cache during the next cycle.

In the first example, `bs` is asserted in Cycle 1 during which `ts` is also asserted. Assuming that `ikill` is not asserted, the instruction in the third delay slot `DS3` is output from the IFU at the end of this cycle. Once this `DS3` instruction has been output there is no need to retain the old `ip` or `fip` so the `ip` is overwritten with the branch destination `B1` at the start of phase 2 of Cycle 2. The sequence shows `ts` also asserted in Cycle 2 so the destination instruction is output. The `fip` is incremented to the next 128-bit *row* of operations at the end of this cycle. The `ip` is incremented by the instruction length (1-3) at the start of phase 2 of Cycle 3.

Cycle 3 shows a branch that is performed when the thread is not selected. This causes a pending branch. The `fip` register is updated immediately. However the `ip` holds the address of the last delay slot and cannot be discarded until the instruction in this delay slot is output.

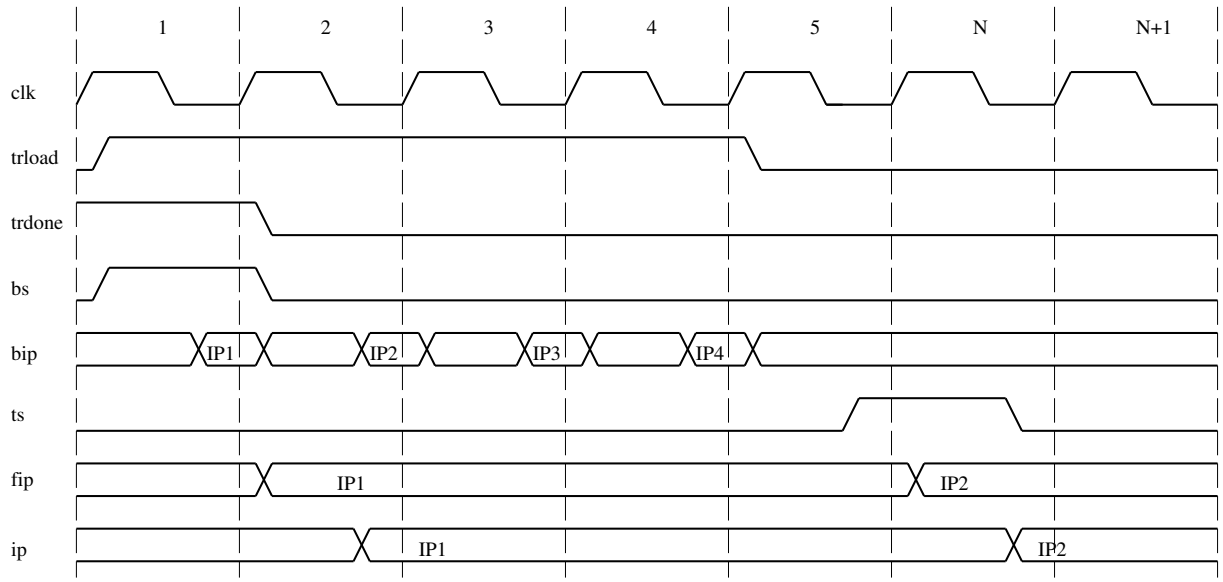

Figure 6: *Thread Restart Sequence*

The `bp` line is asserted to signal this pending condition. The `ts` signal is asserted in Cycle 4 causing the delay slot, `B1+len`, to be output. Once this instruction is output, the old IP is no longer required so in Cycle 5 `bp` goes low and `ip` is updated with the second branch target `B2`.

Cycle 5 shows a branch where the cache misses when fetching the last delay slot. In this case, the `fip` is copied into the `mip` so the miss handler can request the missing line. The branch target address, `B3`, is copied into the `fip` at the end of Cycle 5 and signal `bp` goes high in Cycle 6 to indicate that the `ip` has not yet been updated with this branch target. The miss handling logic generates requests for each word of the missing cache line as indicated in Figure 3 above. When the second word of the requested I-cache row is returned from the memory system, the row is loaded into the operation buffer as shown in Figure 4 above. Once the row is loaded in the operation buffer, the thread becomes available for selection and when `ts` is asserted (Cycle N) the `ip` is updated (Cycle N+1, not shown) just as for the pending branch sequence.

4.6 Thread Restart

Figure 6 shows the sequence used to restart a thread that has been swapped out. To correctly restart a thread that may have been in a branch delay slot when it was swapped out, four instructions pointers must be reloaded into the pipeline. The thread must first be halted by asserting the appropriate `halt` line. Also, a thread restart can only be started when the `trdone` signal is high, indicating that another thread restart is not already in progress.

During Cycle 1 the `trload` signal is asserted along with the appropriate `bs` signal and the first TRIP is presented on the `bip` bus. The branch select signal indicates which thread the

restart is for and trload differentiates this sequence from a simple fork of the thread. On Cycles 2-4 the next three TRIPs are sequenced over the bip lines into the instruction fetch unit by the trload signal being high at the end of the cycle. A dead cycle can be inserted between the loads by making trload low at the end of a cycle.

After all four TRIPs have been loaded the thread may be restarted by negating the appropriate halt signal and waiting for the thread's avail and brok signals to go high. The thread begins execution at IP1 and each of the first three cycles in which the thread's ts is asserted perform implicit branches to the next IP. These branches are handled exactly as described in Section 4.5.

Because the thread restart itself involves a branch as do the first three selections, with ts, of the thread, these operations can only be performed if the appropriate bit of brok is asserted.

5 Detailed Description

5.1 Instruction-Pointer Datapath

The instruction pointer datapath holds all address information within the instruction fetch unit. This includes the IPs, fetch IPs, miss IPs, and thread restart IPs. The unit includes two adders for incrementing FIPs and IPs.

Block Diagram: Figure 7 shows a block diagram of the datapath. Addresses enter on the left on the bip lines. They are gated by a neglatch (NL) and then used to load the fip registers in the event of a branch (blf asserted) and the trip registers (ldtrip asserted). The output of the trip registers is also delayed by a neglatch and used to drive another input of the fip registers that is sampled during thread restart (tlf asserted). An ip register is loaded from the corresponding fip register when signal bli is asserted. The cs signal selects a fip register to be driven onto the fip bus. This bus, after a mux, drives the cache IP cip from which crow and ctag are generated. The mux can be implemented by just tri-stating mip:mtok onto the fip bus as no bit of cs is asserted when mip is selected to drive cip. The fip bus also drives a fip incrementer that increments the selected fip by four and includes an NL stage. The incremented fip is loaded back when ldfip is asserted. Similarly the ts signal selects an ip register to be driven onto the iip bus. An adder, including a PL stage, adds the two-bit len signal to this ip and the resulting value is loaded back if ldip is asserted. On a miss, the fip is loaded into one of the two miss IP, mip, registers when signal ldmip is asserted. The ims and mis signals select one of the mip registers to drive the ima and cip buses respectively during miss handling.

The interface to the IP datapath includes the following signals:

Data Signals:

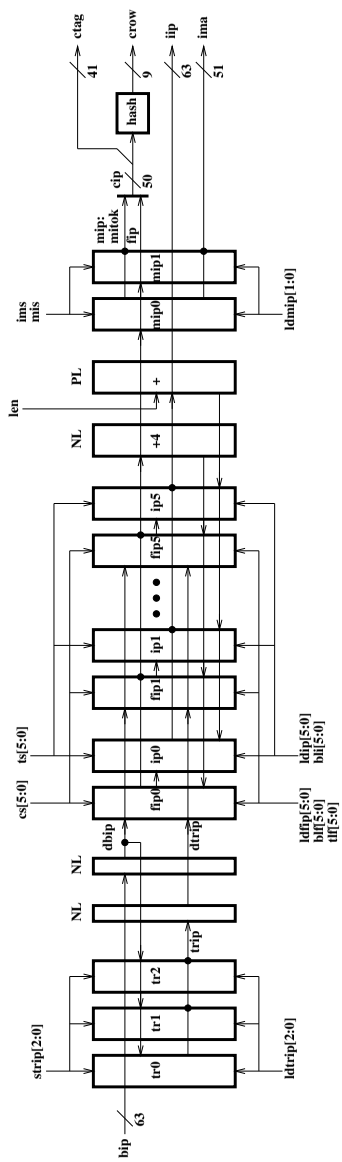

Figure 7: *IP Datapath Block Diagram*

Name	Timing	Dir	Description
bip_v2[62:0]	v2	I	branch IP from top level
iip_1[62:0]	1	O	current IP to top level
illegal_1	1	O	true if current IP has illegal tag
crow_2[8:0]	2	O	cache row select to cache arrays
ctag_2[41:0]	2	O	cache tag to cache tag array
ima[50:0]	B	O	instruction address to M-switch for handling cache misses
len_1[1:0]	1	I	instruction length from OP buffer for IP increment
nxtOps_1[17:0]	1	O	least significant 3 bits of each IP to op buffer
imtok[2:0]	B	I	provides the low 3 bits of IMA
mitok[2:0]	B	I	provides the low 3 bits of CIP during miss handling

Register Load Signals:

Name	Timing	Dir	Description
ldip_q2[5:0]	Q2	I	load IP from IP incrementer output when true
bli_q2[5:0]	Q2	I	load IP from corresponding FIP when true
ldfip_q1[5:0]	Q1	I	load FIP from FIP incrementer output when true
blf_q1[5:0]	Q1	I	load FIP from BIP when true
tlf_q1[5:0]	Q1	I	load FIP from selected TRIP register when true
ldtrip_q1[2:0]	Q1	I	load TRIP from BIP bus when true
ldmip_q2[1:0]	Q2	I	load MIP from FIP bus when true

Select Signals:

Name	Timing	Dir	Description
cs_2[5:0]	2	I	selects the FIP register to be driven on the FIP bus
ts_1[5:0]	1	I	selects the IP register to be driven on the IIP bus
strip_2[2:0]	2	I	selects the TRIP register to drive the TRIP bus
mis	B	I	selects the MIP register to drive CIP during miss handling
ims	B	I	selects the MIP register to drive IMA
cas	B	I	selects between FIP and MIP to drive CIP bus

Figure 8 shows circuit diagrams for some of the cells in the IP datapath. All the circuits use NFET-only pass-gates with low-threshold inverters using a PMOS input pullup. For inverting cells, inverters will need to be inserted or the verilog modified to keep the number of inversions even along all paths. Some of the circuits may be combined with latches (e.g., the adder). Several flavors of registers are used as shown. All are static, have NFET-only input and output switches, and an output isolation inverter. The REG2_1A and REG3_1A cells includes one ungated output and is used for the IPs and FIPs respectively. All of the FIP and the low three bits of the IPs (for `nxtOps_1`) are available directly. The remaining bits of the IPs use the straight REG2_1 cell. To fully restore levels for signals leaving the datapath that have been through NFET-only stages (like `iip` and `ima`, a restoring buffer needs to be used.

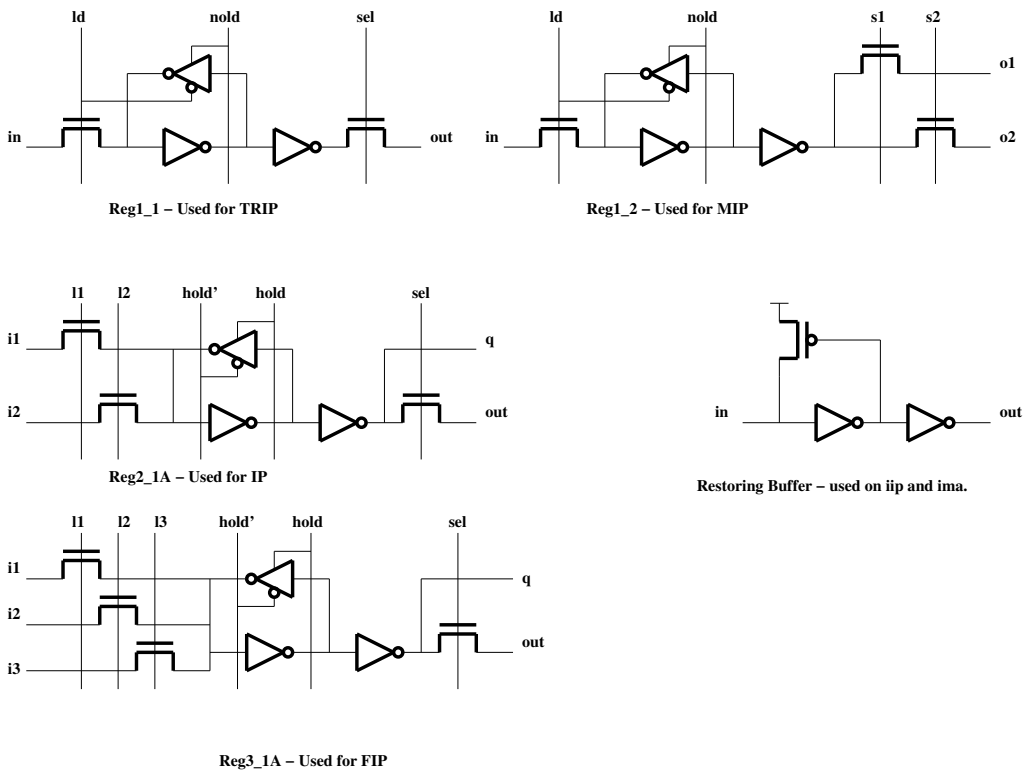

Figure 8: *IP Datapath Cells*

5.2 Instruction-Pointer Control Logic

The IP control logic generates the signals that control the IP datapath.

The following text describes the code section by section:

Buffer Wrap Detection: This logic generates the signals `decAv_1` and `instWrap_1` signals from `iip_1` and `len_1`. `decAv_1` is asserted when the current instruction will consume the last operation in the current line of the operation buffer, and `instWrap_1` is asserted when it consumes at least one operation from the next line of the buffer as well. These signals are used to determine when to decrement the number of available lines in the buffer and when an instruction must be killed because it includes operations from an empty line.

Branch Pending Register: The `bpend` register contains one bit per thread which is set when a thread has a branch pending that has not yet updated the thread's `ip`. When a bit of `bpend` is asserted it disables cache references for the corresponding thread to prevent the `fip` from being incremented before the `ip` is updated. When the thread is selected for issue by `ts`, the `ip` is loaded from the `fip` and the corresponding bit of `bpend` is cleared.

IP Load Signals: The `bli_q2` signal is asserted during clock low to load the `ip` from the `fip` when there was a branch on a halted thread (a fork) or a branch or pending branch to the selected thread at the end of the last cycle. The `ldip_q2` signal is asserted during clock low if no branch is to occur and the thread successfully issued an instruction during the last cycle.

FIP Load Signals: The `ldfip_q1` signal is asserted for a thread during clock high if this thread made a cache reference on the last cycle that either hit or was caught by a miss handler and the thread was not the target of a branch. The `blf_q1` signal is asserted if the thread was the target of a branch during the last cycle.

TRIP Logic: This logic controls the sequencing of the thread restart process. The `trstate` signal holds the current thread-restart state. During state 0, the thread restart sequencer is idle and `trdone` is asserted. When `trld` is asserted the first `trip` is loaded into the `ip` and `fip` if the target thread, indicated by the `trbs` signal, and the state is advanced. During states 1-3, the next three `trip`s are loaded into the three trip registers on each cycle in which `trld` is asserted. After the loading is complete the sequencer generates fake branches in states 4-6 each time the thread is selected. The `trms` signal switches the `trip` signal onto the `dbip` bus during these states (hence no branches by other threads are allowed during thread restart). Each selection of the restarting thread generates a fake branch on the `tbs_v2` signal and advances the state which in turn selectes the next `trip` via the `strip_2` signal. Because the cache arbiter needs the `bs_2` signal early in the cycle, the `tbs_v2` signal cannot

be used to drive the arbiter. The `tcbs_2` signal is generated to replace this signal and gives cache preference to the restarting thread.

Circuits: The only critical circuits in the control logic involve the generation of the qualified clocks. This cannot be left to synopsys. Each qualified clock drives about 64 passgates. Appropriately sized CKA or NCKA cells should be used to drive these signals.

5.3 Instruction Cache and Control

Data Array: The I-cache data array is a 512x128 RAM that is to be implemented as a 256x256 array with two bit-line pairs multiplexed into each sense amplifier. The 128-bit output is to be interleaved at the operation (32-bit) level. The RAM is writeable in 64-bit chunks (also operation interleaved). During a write, the data output lines must provide valid data for the entire selected line. The write data lines to the array must also be interleaved at the operation level. This will require a 32-track crossover to interleave the high- and low-half words of the write data.

Tag Array: The I-cache tag array is a 128 by 43 (42 tag plus 1 valid bit) RAM array that is to be implemented as a 64 by 86 array with two bit-line pairs per sense amplifier. The array includes a comparator built into the bit-line peripheral circuits that asserts `match_v2` when the addressed word matches `tin` and the valid bit is set. The last two columns (the valid bits) use a special 7-transistor RAM cell that includes a master clear to set all of the rows to the invalid state on a reset.

Array Circuits: The array circuits are described in the document: *MAP RAM Circuit Design.*

Miss Kill Logic: This logic determines when to assert the `kill` signal. It separately determines if an I-cache miss causes a kill because the current instruction is from an otherwise empty buffer (`missKill`) or if the current instruction wraps into a buffer that will not be full at the end of the cycle (`wrapKill`. If either of these conditions holds, `kill` is asserted.

Cache Arbiter: This logic determines which thread gets access to the cache on a cycle-by-cycle basis and outputs the selection on the `cs_2` signals. When the miss handler is writing to the cache, the `cbusy` line is asserted and no thread is allocated the cache. Otherwise, first priority goes to a branching thread with an empty buffer to fetch the last delay slot before the branch overwrites the `fip`. (The `bs` signal here is really `tcbs` from the IP control logic.) Second priority goes to threads with empty buffers, both `nav` signals low. Third priority goes to any thread without a pending branch, without a pending miss, and with room in its op buffer. Among each class priority is fixed.

5.4 Miss Handler

The miss logic consists of a miss register that records which threads have pending misses and two miss handlers. Miss handler 0 deals with threads 0-4, the normal and exception threads while miss handler 1 deals with thread 5, the event thread. This is critical on Cluster 1 to prevent the LTLB handler from being blocked by other thread's I-cache misses. The miss handlers control the `mip` registers in the IP datapath.

For miss handler 0, once one of the five threads misses, the other threads are prohibited from branching - to prevent the `fip` from being overwritten when a `mip` is unavailable to capture the state.

On a miss, the missing address is captured in the appropriate `mip` register by asserting the `ldmip_q2` signal. This signal is actually asserted on every cycle in which the miss handler is idle as the notice of a miss comes too late to initate the signal. The low two *row* bits of the missing IP (bits 4 and 5) are redundantly stored in the `missIp` register. This register is added to the request counter in order to request the needed row first. The missing thread is captured in the `missThread` register.

The state of the miss handler is contained in two one-bit flags, `req` and `rec`, and two three-bit counters, `mcnt` and `rcnt`. Initially all are zero. When a miss is detected, the `start` signal is asserted and the `req` flag goes high on the next cycle. While `req` is high, the unit requests memory accesses by asserting `imreq` and advancing `mcnt` each time the m-switch signals `imack`. The counter, along with a 1-bit tag identifying the fault handler forms a token, `imtok`, that identifies the request.

As words return from the memory system, signal `mireq` is asserted. This causes the word to be written to the I-cache at the location specified by `mitok` and the appropriate `mip` register. The first time the returning token matches the `missIp` register, the `rr` (requested row) register is set to indicate that the first half of the requested row has been fetched. The second time the token matches `missIp`, the `missld` signal is asserted to load the entire requested row into the operation buffer and increment the thread's `fip`.

When the last request is acknowledged, the `req` signal goes low and the `rec` signal is asserted. This indicates that all requests are complete and the miss handler is in the *recieve* state. Each returning word causes the `rcnt` register to be incremented. When the `rcnt` register indicates that the last word has returned from memory, the `rc` or `finish` signal is asserted and the handler returns to its idle state. The handler must spend at least one idle cycle between requests in order to properly capture the state of the miss.

The `IF_MISS_BANK` module combines the two miss handlers and includes logic to arbitrate between them giving fixed priority to handler 1. The miss bank also includes the `brok` logic which asserts a *branch OK* signal for each thread when the thread's miss handler is idle and the thread's operation buffer is not empty. This assures that the state required to fetch the last delay slot before the branch target will be retained. It is up to the SZ stage to make sure that no branches occur during a thread restart.

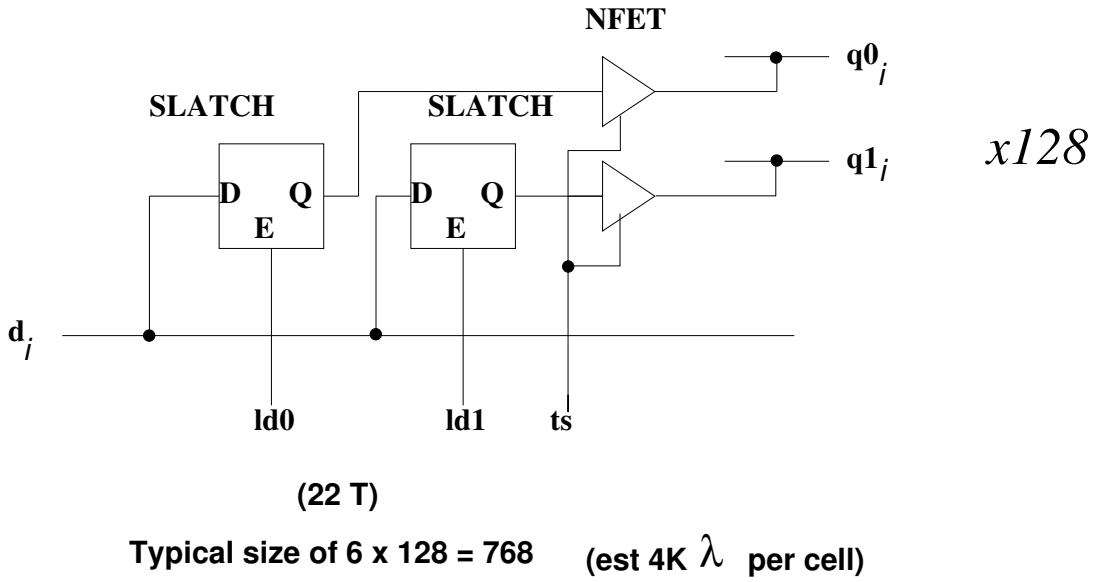

Figure 9: *Operation Buffer Cell*

5.5 Operation-Buffer Datapath

The operation buffer stores operations and forms them into instructions according to the two *pack* bits per operation. Six 256-bit operation buffers, one for each thread, are split into two 128-bit halves that are loaded as a unit from the instruction cache. Each 128-bit half-buffer contains four bit-interleaved 32-bit operations. Thus, the MSB (bit 31) of all four operations are next to each other at the left end of the operation buffer. This interleaving greatly simplifies the multiplexing required to delimit instructions and select operations.

Figure 9 of the Operation buffer drawings shows the basic buffer cell. This cell contains two static latches (one bit for each of the two half buffers) and two NFET pass gates. As shown on Figure 10, these cells are composed in a 128x6 array. The inputs of the array are the 128 data lines from the cache while the outputs are 256 lines representing the entire state of the buffer for the thread selected by *ts*

The 256 bits out of the array of cells connects to the inputs of 32 3→8 selectors (*swizzler*) as shown on Figure 10. The schematic for these selectors is shown on Figure 11. The 0,4,1,5,...,3,7 ordering of inputs due to the buffer arrangement may complicate layout slightly. These selectors select the next three 32-bit operations (if they are in the buffer) from the selected thread's instruction stream. To this end, they are driven by the NOPD[0:7] signal, the decoded *nxt_op* register of the selected thread.

The upper two bits of the operation buffer hold the "PACK" bits that specify how the operations are packed into instructions. The pack bits for the current and next operation out of the two most significant 8→3 selectors are called CUR_PACK and NEXT_PACK. The third output from these two selectors is unuused. The "PACK" bits are used to calculate:

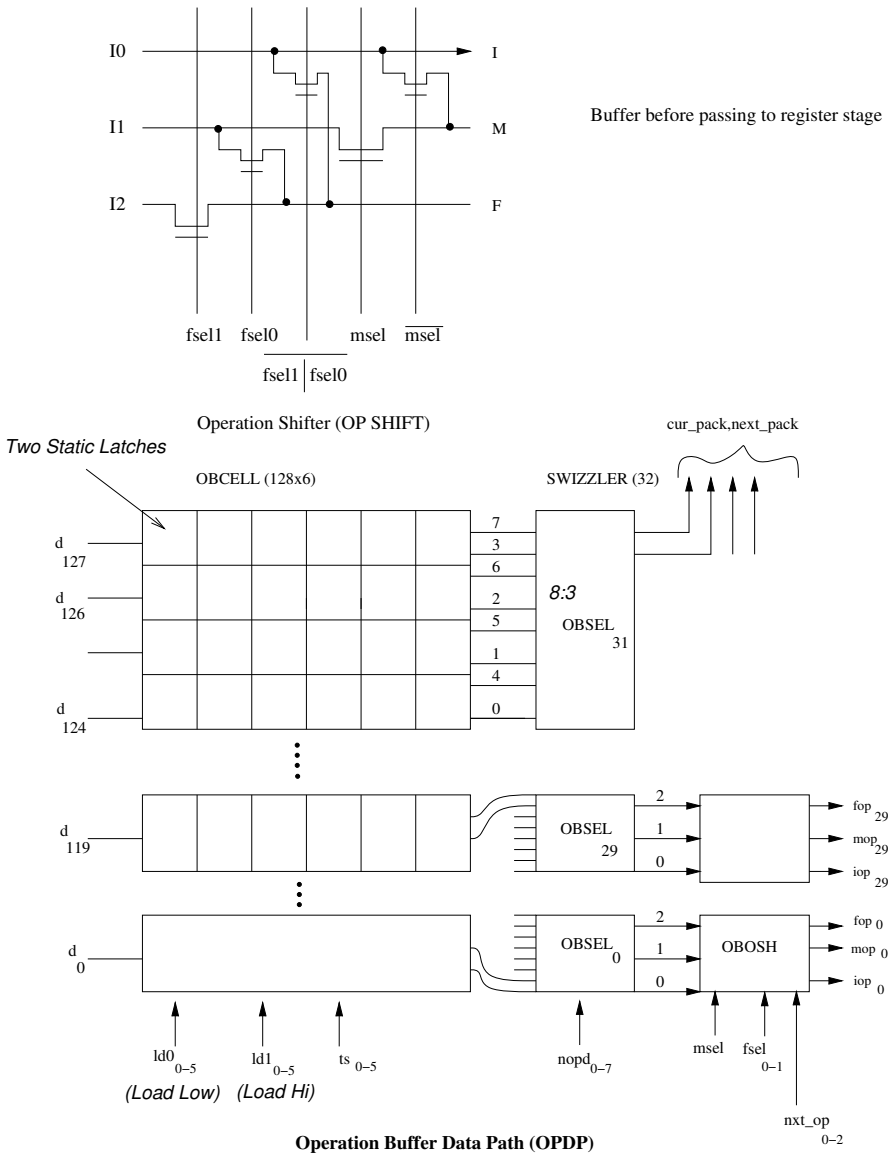

Figure 10: *Operation Datapath*

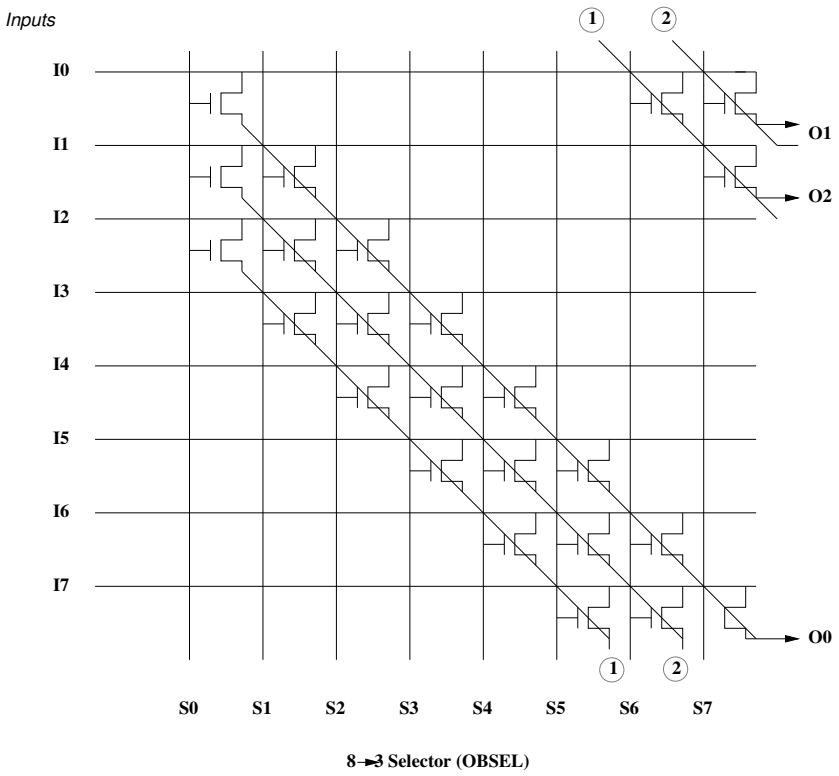

Figure 11: *Operation Buffer Selector (Swizzler)*

instruction length (LEN), whether the instruction is valid (do we have enough ops in the buffer), if this instruction includes I, M, and/or F ops, and if so where the I, M, and F ops are.

Then the presence of each of the three operations (IV, MV, and FV) is determined from the PACK fields and their positions are calculated. The positions are encoded in a 5-bit OPSH (operation shift) signal that is used to drive the 30 operation shifters at the end of the data path. The output of the shifter is the IOP, MOP, and FOP signals. These signals are buffered and sent to the register (RR) stage of the pipeline.

5.6 Operation Buffer Control

Instruction Length Logic: The *pack* fields from the next two operations, `cp_1[1:0]` and `np_1[1:0]`, are used by the instruction length logic to compute the instruction length, `len_1[1:0]`, the select signals for the shifter, `msel_1[1:0]` and `fsel_1[2:0]`, and the operation valid signals, `iv, mv,` and `fv`. This is strictly combinational logic and is in the critical path from `ts_1` to operations out.

Next Operation Decoder: This 3 to 8 decoder generates the one-hot `decNxtOp` signal from the low three bits of the `iip`. The decoded signal is used by the swizzler to select the next three operations from the selected thread's buffer.

Number Available Logic: An instance of module `IF_AVAIL` is instantiated for each thread to keep track of the number of valid rows a particular thread has in its operation buffer. This number is kept in the `nrAvail_2` register and is output on the `nav1` and `nav0` signals that are combined with the corresponding signals from the other `IF_AVAIL` blocks to form six-bit (one-per-thread) signals.

The `nrAvail` count is zeroed on reset or on a successful branch (`bs` or `bp` true when `ts` is asserted and `kill` is not). Otherwise, it is decremented if the current instruction consumes the last operation of the current buffer (as signalled by `decAv_1`) and incremented on a cache hit or miss load.

This logic may need to be hand tweaked to keep the paths from `ts`, `kill`, and `match` to the register short.

Operation Buffer Load Logic: This module generates the 12 load signals for the operation buffer. Each signal is asserted when the appropriate bit of `cs` or `missld` is asserted and the low bit of the cache IP is in the appropraite state. The signals are qualfied by clock low.

A Overview Diagram

An overview of the IFU control and datapath logic is provided in Figure 12.

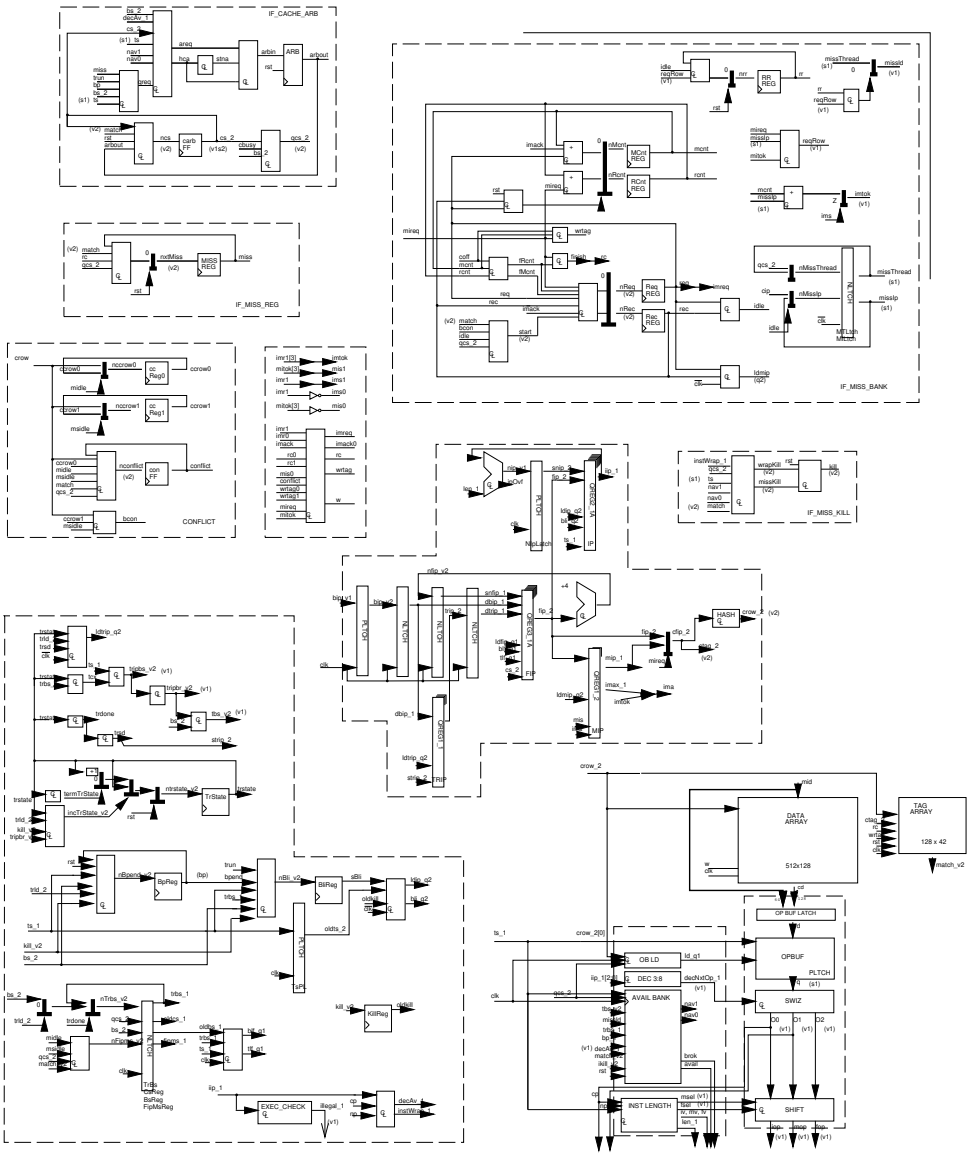

Figure 12: *Logic Design Overview*

B IFU Datapath Timing Specifications

Table 1: IP Input Constraints

Item	Min	Max	Notes
ts_1: Data setup from clk↑ (t_{TSSU})	**X**		
len_1: Data setup from clk↑ (t_{LNSU})	**X**		
cs_2: Data setup from clk↓ (t_{CSSU})	**X**		
strip_2: Data setup from clk↓ (t_{STSU})	**X**		
ldfip_q1: Data setup from clk↓ (t_{Q1SU})	**X**		
blf_q1: Data setup from clk↓ (t_{Q1SU})	**X**		
tlf_q1: Data setup from clk↓ (t_{Q1SU})	**X**		
ldip_q2: Data setup from clk↑ (t_{Q2SU})	**X**		
bli_q2: Data setup from clk↑ (t_{Q2SU})	**X**		
ldtrip_q2: Data setup from clk↑ (t_{Q2SU})	**X**		
ldmip_q2: Data setup from clk↑ (t_{Q2SU})	**X**		
imtok: Data setup from clk↓ (t_{IMSU})	**X**		
mis,ims: Data setup from clk↓ (t_{IMSU})	**X**		

Table 2: IP Output Timing

Item	Min	Max	Notes
iip_1: Data Valid from clk↓ (t_{IPV})		**X**	
iip_1: Data Valid from ts_1 (t_{IPTS})		**X**	
crow_v2: Data Valid from clk↑ (t_{CRV})		**X**	
crow_v2: Data Valid from cs_2 (t_{CRCS})		**X**	
ctag_v2: Data Valid from clk↑ (t_{CTV})		**X**	
ctag_v2: Data Valid from cs_2 (t_{CTCS})		**X**	
ima: Data Valid from clk↑ (t_{MAV})		**X**	

NOTE(S):

1. Please refer to Figure 13 for the input constraint timing.
2. Please refer to Figure 14 for the output timing.

I-Cache Test Procedure

The I-Cache data and tag arrays are tested from the pins of the IFU according to the following procedure. Two pins `tstrd` and `tstwr` are provided to activate these test functions.

To Load the Data and Tag Arrays:

1. Disable execution by setting `trun` to zero for all threads.
2. Set the address to be loaded by asserting the address on `bp` and asserting `bs[0]`. This loads the address into FIP0.
3. Write data into the RAMs by placing the data on `mid` and asserting `tstwr`. This writes `mid` into the data RAM and writes the high bits of FIP0 into the tag RAM both at the row selected by the low bits of FIP0. As a side effect, this increments FIP0.
4. Repeat steps 2 and/or 3 as needed to complete loading of the RAM.

To Read Data out of the Data and Tag Arrays

1. Set the address to be examined by asserting the address on `bp` and asserting `bs[0]`. This loads the address into FIP0.
2. Read the data into the output shift register by asserting `tstrd`. This loads the 128-bit row into the output shift register and captures the `match` signal. As a side effect, FIP0 is incremented.
3. Shift out the data and match signal.
4. Repeat steps 2 and 3 as needed to examine the remainder of the RAM.

This procedure depends on external circuitry to provide addresses on the `bp` lines and data on the `mid` lines.

Addenda

A Faults on I-Cache Misses

When a miss handler issues a request to the memory system that requires lengthy software handling it is necessary to free up the miss handler without completing the operation.

To this end, modifications will be made to the IFU control so that cache misses are handled as follows:

1. On a cache miss the FIP will not be incremented. This differs from the current configuration where the FIP is incremented if the miss is caught by a miss handler, but not otherwise. The current design is intended to avoid a livelock problem during miss conflicts. We will deal with this problem in another manner.

2. If the requested row (two words) is returned by hardware it is bypassed (via the recent op-buf mod) into the op buf and the "nav" (number available) count for the thread is incremented. A new per-thread fetch-ahead flag "fa" is also set in this case to prevent double fetching of the word.

3. When the thread is next selected for a cache reference (after the miss handler clears the miss register) the FIP is incremented and the "fa" bit is cleared; however no data is loaded into the op-buf and the "nav" state is unmodified. The high "fa" inhibits these operations. After this cycle the FIP, IP, and op-buf are again in a normal state (rather than the "fetch ahead" state).

4. If the entire requested row is not returned by hardware, "nav" will not be incremented and "fa" will not be set. In this case, software must halt the thread and for each word that is not returned by hardware, software must perform a config space write to a special location that advances the miss handler. Upon the completion of these writes the miss handler is free to handle other requests and the state of FIP and "nav" is as if the fetch had never occured.

INPUT CONTRAINTS

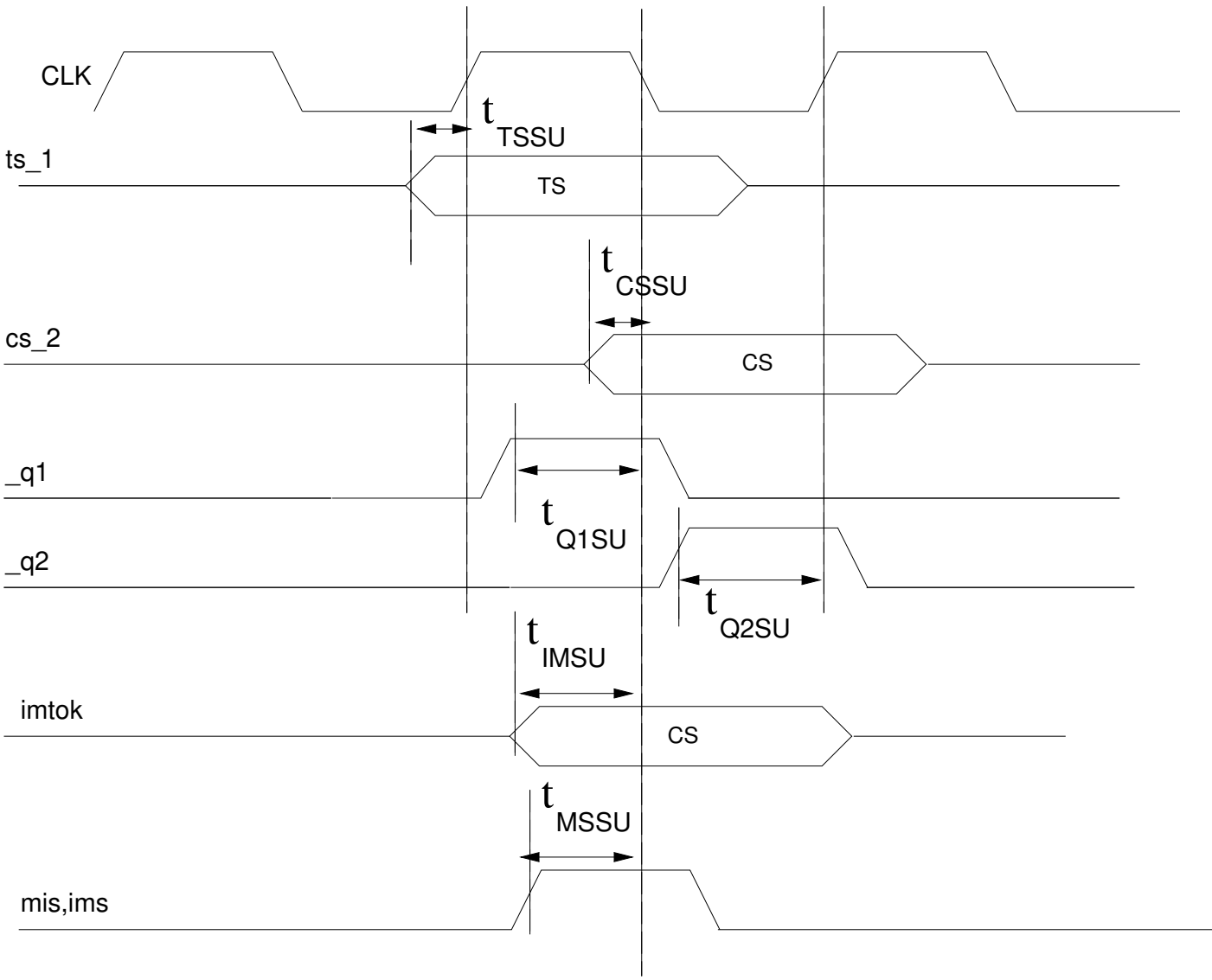

Figure 13: *IP DP Input Specification*

OUTPUT TIMING

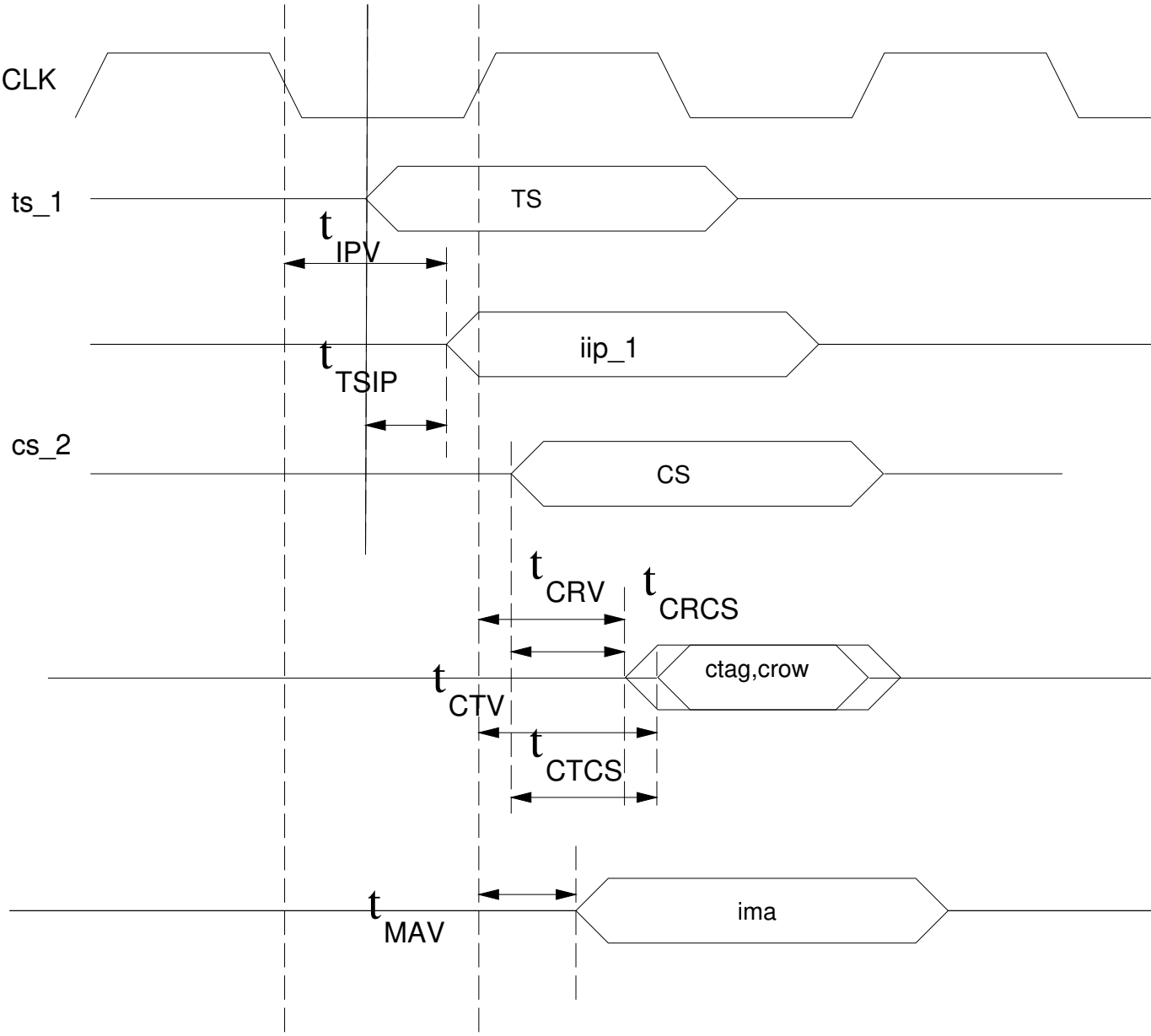

Figure 14: *IP DP Output Specification*

Table 3: OPBUF Input Constraints

Item	Min	Max	Notes
d: Data setup from ld↓ (t_{DSU})	**X**		

Table 4: OPBUF Output Timings and Propogation Delays

Item	Min	Max	Notes
op_1: Data Valid from ts_1↑ (t_{TS2OP})		**X**	iop, mop, fop
op_1: Data Valid from ld↑ (t_{LD2OP})		**X**	iop, mop, fop
op_1: Data Valid from d valid (t_{D2OP})		**X**	iop, mop, fop
op_1: Data Valid from decNxtOp_1↑ (t_{DC2OP})		**X**	iop, mop, fop
mop_1: Data Valid from msel↑ (t_{MS2OP})		**X**	
fop_1: Data Valid from fsel↑ (t_{FS2OP})		**X**	
cp: Data Valid from decNxtOp_1↑ (t_{DC2CP})		**X**	
np: Data Valid from decNxtOp_1↑ (t_{DC2NP})		**X**	

INPUT CONTRAINTS

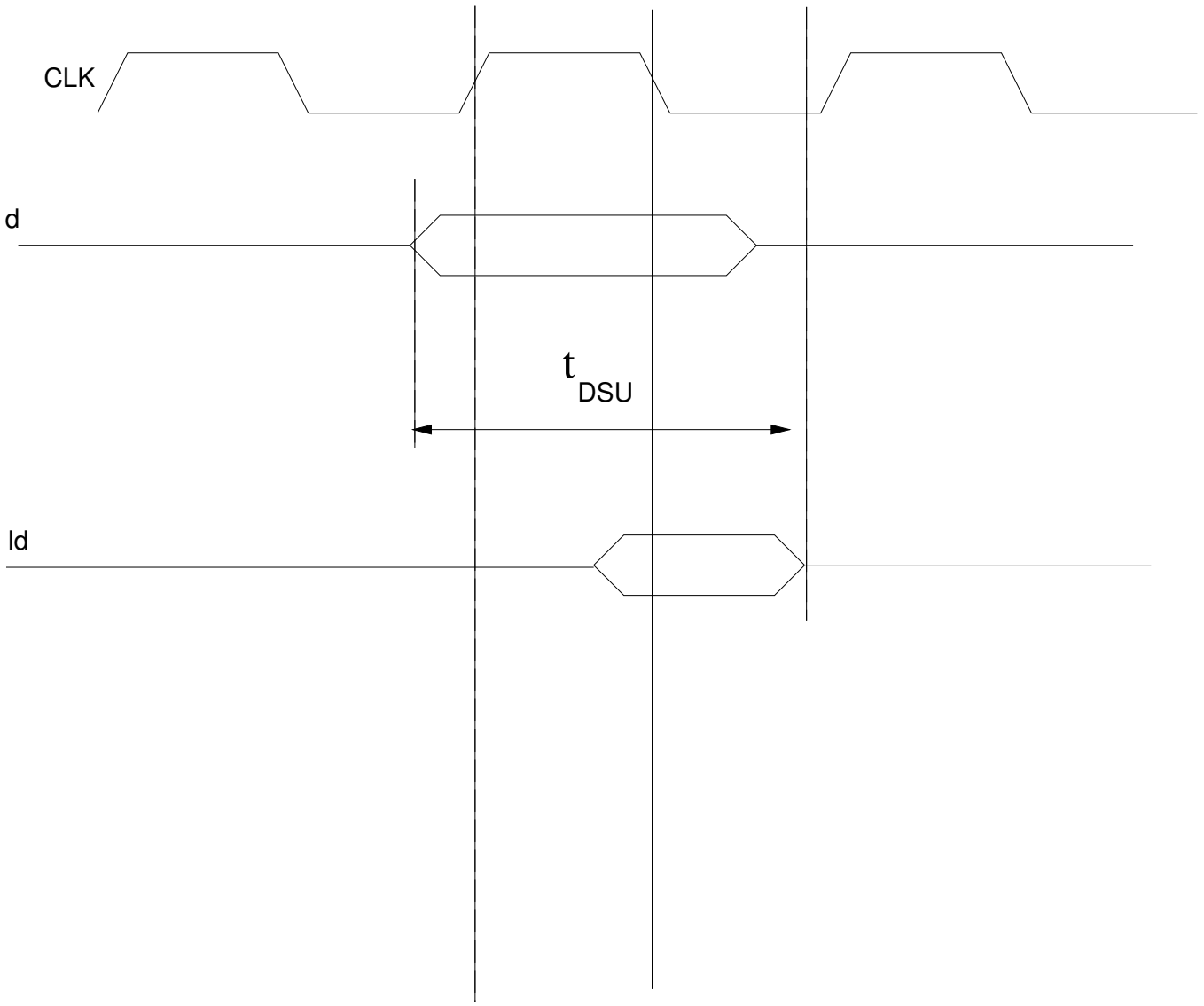

Figure 15: *OPBUF DP Input Specification*

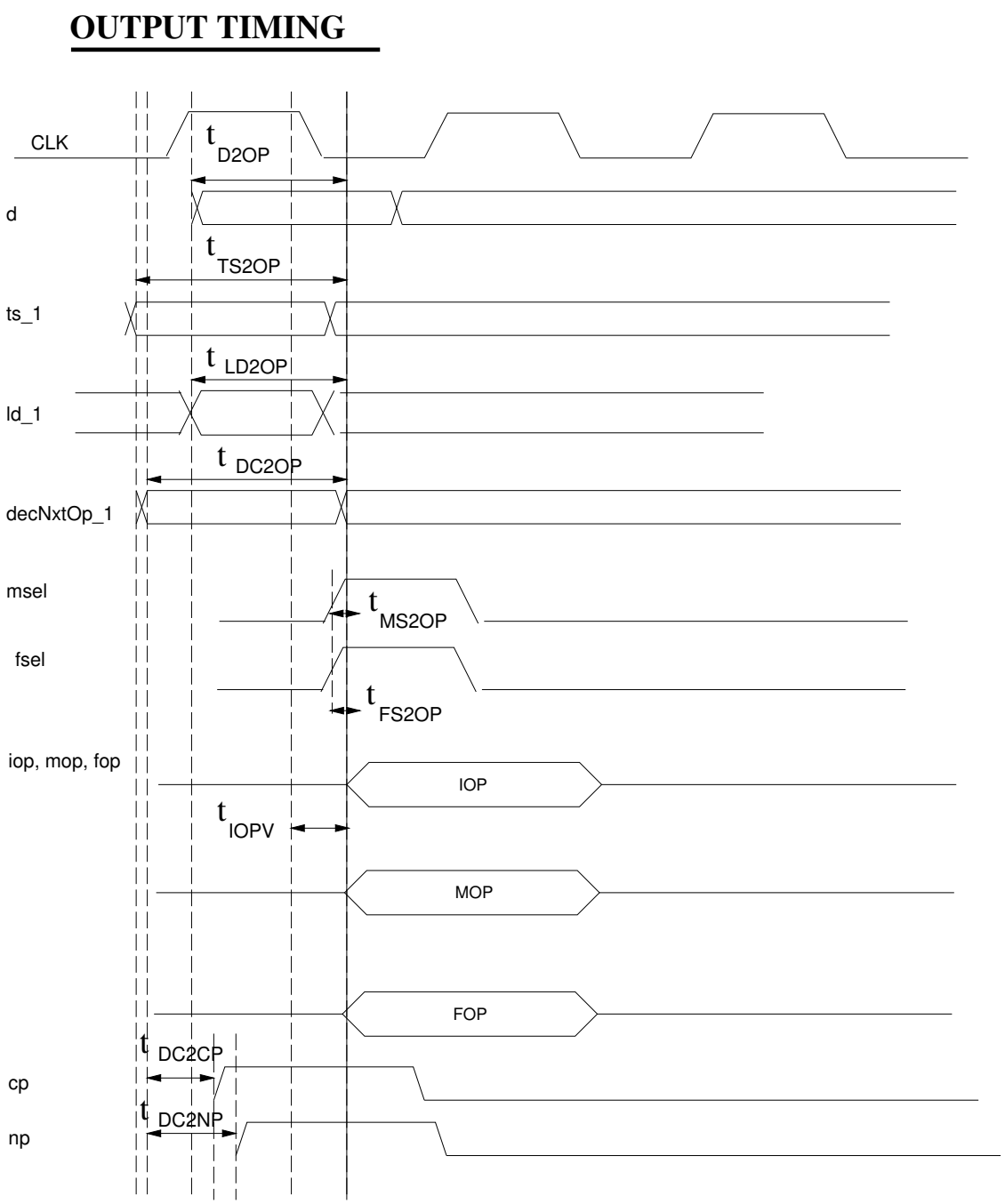

Figure 16: *OPBUF DP Output Specification*