PPE 42 & 42X Embedded Processor Core

Microarchitecture

Version 2.5

November 15, 2019

IBM Corporation
Systems & Technology Group
11400 Burnet Road
Austin, Texas 78758

c/o Michael Floyd, mfloyd@us.ibm.com

Disclaimer

This document was a working document during development of the PPE 42X core microarchitecture. It may no longer be accurate or reflect late changes in the design. The definitive microarchitectural specification is now the semi-formal specification found at the following URL:

https://ibm.box.com/ Link for PPE 42X Core Users Manual.pdf

Source Documents

The latest version of this document may be found at:

https://ibm.box.com/ Link for PPE 42X Core Microarchitecture.pdf

The source files for this document can be accessed via the following URL:

https://ibm.box.com/ Link for PPE Microarchitecture folder

Note that change bars for sub-documents of Open Office master documents are lost when Open Office master documents are rendered into PDF. In order to view change markings of sub-documents is it necessary to view the Open Office sub-documents directly.

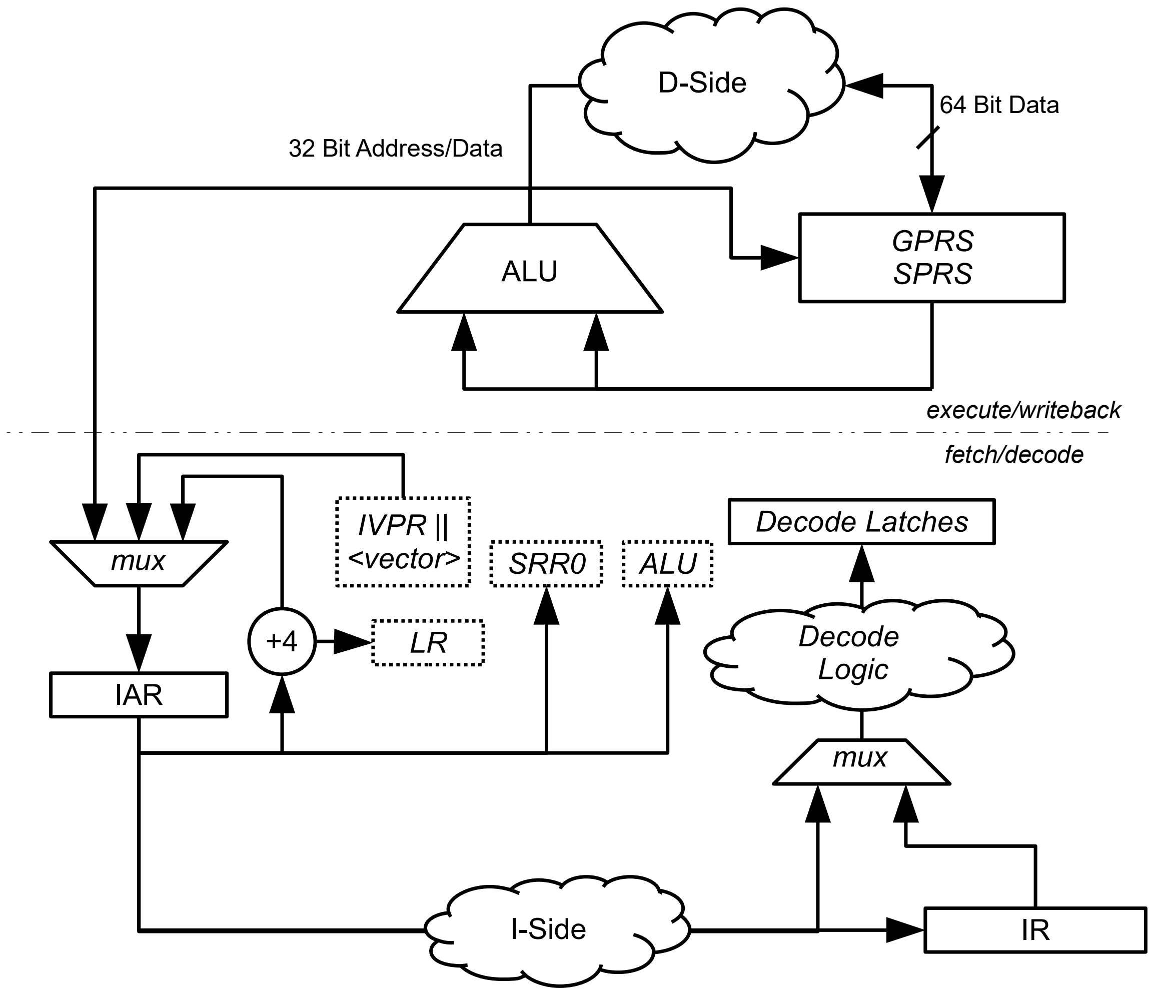
Revision History

Version	Date	Changes	Author
2.5	11/15/2019	• Added table to better summarize state encodings and description summary.	
2.4	03/29/2019	• Added a few clarification comments on reset and force halt behavior	mfloyd
2.3	03/07/2019	• Corrected Stack state to not update RA from memory as part of VDR loads, such that final RA computation in the serial state is correct • Corrected Frame state definition of *n* to include the final word operation to memory (previous version was off by 1)	mfloyd
2.2	02/13/2018	• Corrected Class K alignment check in data state table	mfloyd
2.1	02/08/2019	• Added check for RA alignment and a few corrections to simplify	mfloyd
2.0	01/23/2019	• Updated to reflect PPE 42X changes & additions	mfloyd
1.4	02/19/2015	• Clarified asynchronous (including external interrupt) exception processing.	mfloyd
1.3	07/22/2014	• Corrected the initial block diagram • Changed DBSR to ISR • Many edits for specification changes occurring over the last several months.	Bishop Brock
1.2	4/10/2014	• Renamed Class T (Move To) to Class M; Added Class T (trap) • Added column showing clearing condition of hidden state • Reduced the amount of hidden state • State table revisions as discussed in review meetings • Other updates from review meetings	Bishop Brock
1.1	04/04/2014	• Modified definition of the halted state to include not having a pipelined instruction. • Added comments about IR update • Added comments about how IR gets into EDR • Minor editorial changes	Bishop Brock
1.0	04/02/2014	First version of the final plan-of-record detailing that the core will be implemented as a simple 2-phase model.	Bishop Brock

Table of Contents

1 Introduction

Illustration 1: PPE 42 Core: Very High-Level Block Diagram



The illustration on the preceding page is a very high-level block diagram of the PPE microarchitecture.

On the fetch/decode side, the IAR drives out through the I-side of the memory interface, gets an instruction and decodes it in a single cycle. The IR is mainly used for ramming, however we continue to update the IR during normal fetch since if there is a program exception we need to hold the IR until interrupt prioritization, so that it can be copied to the EDR. The IR is written with the instruction returned from the I-side only when the I-side ACKs an instruction fetch request. The IR can also be written directly to initiate ramming.

The IAR is supported by a dedicated incrementer. The IAR is incremented only when non-branching instructions complete. The IAR feeds the ALU to support branch computations. Branches are handled by writing the IAR with the branch target if the branch is decided as taken.

Since the interrupt vector computation is a simple concatenation, and there is otherwise no need for the vector arguments in the data path, the interrupt vector is directly loaded into the IAR to make the interrupt branch. The IAR is also written directly to SRR0 when an interrupt branch occurs.

On the execute/writeback side, GPRs/SPRs drive the ALU and back into the GPRs/SPRs in 1 cycle. For data operations, the address computation (ALU adder) is fast-pathed through the data side to support loading data back into the GPRs in a single cycle. The ALU provides 32 bits for GPR/SPR update, as a data address, and for new IAR computations for branches. The D-side provides a 64-bit interface to the GPRs for loads and stores.

Note that as a simplification, I-side and D-side transactions are mutually exclusive. The PPE core can sustain CPI = 1 for arithmetic and compare instructions with an infinite single-cycle I-side. The PPE core can sustain CPI = 2 for load/store/dcb* in the presence of an infinite single-cycle D-side.

2 Architected State

The following table details the PPE 42 core architected state. This is the PPE 42 core state that is visible to PPE 42 programs, and to external users as External Interface Registers (XIRs).

Table 1: PPE 42 Register State

Register(s)	This Register is Driven by	This Register Drives	Notes
GPRs	ALU	ALU	GPRs can be considered as a register file with 4 32-bit read ports and 3 32-bit write ports. The read ports are used for arithmetic, “move-to” and stores. Four read ports are required to support **stvdx**. One write port is used for arithmetic, “move-from” and update-form addressing. The other 2 write ports are used for loads. The architecture guarantees that the 3 write addresses are always unique.
CR, XER	ALU	GPRs Control Logic	CR and XER are tightly coupled with ALU operations, and can also be read and written as SPRs. CR controls branching.
CTR	ALU Self (-1)	ALU Self (-1)	CTR is a decrementer with a special (CTR == 1) circuit used to decide branches involving CTR. CTR is involved with branching and feeds the ALU to form the instruction address for the **bcctr** instruction. On PPE42X, CTR is also read-only as an XIR.
LR	ALU IAR (PPE42X: Memory Interface)	ALU	LR is involved with branching and has a special direct connection to be loaded from the IAR (+4) for branch-and-link. On PPE42X, the LR is loaded directly from the memory interface during execution of the **lsku** instruction. On PPE42X, LR is also read-only as an XIR.
DACR	ALU	ALU Control logic	DACR contents are compared with instruction and data addresses to signal debug events.
DBCR	ALU	ALU Control logic	DBCR controls debug event handling. All debug events halt the processor.

Continued on next page

Table 2: PPE 42 Register State (Cont.)

Register(s)	This Register is Driven by	This Register Drives	Notes
DEC	ALU Timer inputs	ALU TSR	DEC decrements on every cycle the selected decrementer input is '1'. DEC is read and written as a normal SPR. When DEC_0 transitions from 0 → 1, TSR[DIS] is set.
EDR	ALU IR Memory Interface	ALU External	EDR is loaded from the ALU by **mtspr**. EDR is loaded directly from the IR for program exceptions to facilitate debug. EDR is loaded directly from the memory interface to provide the failing data address for data machine check, data storage and alignment interrupts. EDR is also read-only as an XIR. On PPE42X, EDR is loaded from the ALU, and is a source into the ALU to form data addresses, during execution of the **lsku** and **stsku** instructions.
IAR	ALU Self (IAR + 4) IVPR || <vector>	ALU SRR0 External Self (IAR + 4)	During execution the IAR is always the NIA. When halted the IAR always points to the instruction to execute next once processing resumes. To simplify exception handling SRR0 is loaded directly from the IAR, and IAR is loaded directly with IVPR || vector. IAR is also read-only as an XIR.
ISR	Control logic	ALU Control logic	Bits in ISR are set during interrupt processing.
IVPR	External	ALU IAR	Concatenated with the interrupt vector offset to form the address to fetch during interrupt processing.
MSR	ALU Control logic Memory Interface	ALU Control Logic Memory Interface	MSR has many special functions and connections. MSR bits are cleared during exception handling, based on the exception. MSR[SEM] and MSR[IPE] are presented to the memory interface. MSR[SIBRC] and MSR[SIBRCA] are updated by the memory interface. MSR bits ME, UE and EE control exception processing. MSR[EE] has a special update from the RDR to implement **wrtee** and **wrteei**.
PIR	External	ALU	A read-only SPR to the core; identifies the specific PPE instance.
PVR	Constant	ALU	A generic constant, read-only to the core.
SPRG0	ALU External	ALU External	SPRG0 is read/write as an XIR.
SRR0	ALU IAR	ALU	SRR0 has a special connection that allows it to be loaded directly from IAR during exception processing. **rfi** decodes similarly to a “branch-to-SRR0” so in this case SRR0 moves through the ALU. On PPE42X, SRR0 is also read-only as an XIR.
SRR1	ALU MSR	ALU MSR	SRR1 moves through the ALU for **mfsrr1** and **mtsrr1**, however for exception processing and **rfi** SRR1 is loaded from and written to the MSR directly.
TCR, TSR	ALU (Special) Timer inputs	ALU Control logic	Timer control and status. DEC, FIT and WDT events set bits in TSR. TSR implements special write-1-to-clear semantics for status bits, however these bits are directly writable while ramming.
XCR, XSR	External Control Logic	External Control Logic	XCR contains bits controlling reset, halt, single-step and RAM and is writeable as an XIR. XSR mirrors DBSR, ISR, and other status bit externally, readable as an XIR. On PPE42X, a subset of the XSR is also writeable as an XIR.

3 Hidden State

Due to the pipelined nature of the PPE 42 core, a great deal of non-architected state is required, especially the ALU decode information. Proper interrupt prioritization and handling requires there to be a bit of state for most of the interrupt types. This hidden state is not cleared until after interrupts are prioritized.

The table below summarizes the type of hidden state required to implement PPE 42 core state machine.

Table 3: PPE 42 & 42X Core State Machine Hidden State

State	Associated With or Recognized During	Clearing Condition (If applicable)	Notes
ALU operation	**home**	It should not be necessary to clear all decode latches, as long as any “action” latches are cleared when the action takes place.	Decode latches that record ALU input sources and operations. All ALU function is controlled by these latches; there are no special-case or fast-path overrides.
Writeback operation			Decode latches that record the writeback operation. These latches are set during instruction decode and do not change.
Instruction Type			There are several categories of instructions with different processing requirements. These latches are set during instruction decode and do not change.
pipelined		Always cleared the cycle the action is taken, unless it is also set on that cycle.	The previous fetch decoded as a pipelined arithmetic or branch, which is completed concurrent with the current fetch.
Instruction Machine Check Event		**vector -----> home**	Instruction machine checks are recognized in the **home** state. ISR and EDR are updated immediately, but may be overwritten.
Instruction Storage Event		**vector -----> home**	ISI events are recognized in the **home** state. They may get prioritized out so the hidden state is needed.
External Event		**vector -----> home**	The external event is indicative, but not definitive. For numerous reasons, the external input can glitch. Therefore it would be possible to begin interrupt processing with an external input event present, but find the external status has disappeared at the end of synchronization. In general this can never be avoided except by careful programming. Therefore this bit is latched each time a check is made for pending asynchronous exceptions, and maintains latched until hidden state bits associated with exceptions are cleared after prioritization. Note that the DEC, FIT and WDT status are safe and can not glitch, only the external input can glitch. Therefore we don't need hidden status for the timer interrupts as their presence is easily computed by a simple AND of their status and enable.
Program Event		**vector -----> home**	The program exception is recognized and handled during **home.** We need hidden state bits because we can't update ISR until and unless we actually take the program interrupt after prioritization. We can set ISR[PTR] from these two bits. Note we can update EDR from the instruction interface if a program event happens, and if synchronization uncovers a machine check it will just get overwritten.
trap Event			

Continued next page

Table 4: PPE 42 & 42X Core State Machine Hidden State (Continued)

State	Associated With or Recognized During	Clearing Condition (If applicable)	Notes
Data Machine Check Event	**data**	**vector -----> home**	Data machine check events are recognized and handled in the **data** state. ISR and EDR are updated immediately.
Data Storage Event			DSI events are recognized and handled in the **data** state
Alignment Event			Alignment events are recognized and handled in the **data** state
Load/Store indicator			Data storage and alignment need a load/store indicator for ISR, which must be held for prioritization.
System Reset Event	**reset**		Used to prioritize interrupt handling.
Hard Reset			Indicates whether a system reset is “hard” or “soft”
Number of Stack Transactions	**frame** **stack**	**stack -----> sync** **stack -----> serial**	4-bit incrementer that maintains a count of how many memory transactions have been performed by Class K instructions in the **stack** state. This hidden state is initialized to 1 in the **frame** state, and is cleared to 0 when leaving the **stack** state.

4 Instruction Handling

The majority of instructions require no special handling, and are pipelined with a maximum completion rate of CPI = 1 for arithmetic, compare, **mfspr**, **mtcr0** and **mfcr0**, and CPI = 2 for data operations not including memory delay. Fused compare-branch instructions have a CPI =3, whereas stack frame processing instructions, new to PPE42X, have a CPI of between 4 and 13 not including memory delays. The table below details at a high level how each class of instructions is handled.

Note that cycle times for instructions requiring either form of synchronization delay include a minimum of 1 cycle for generating a synchronization transaction. Extra synchronization delay is only required if the memory interface does not respond to the synchronization request in 1 cycle.

Table 5: PPE 42 & 42X Core Instruction Handling

Instruction Class	Instruction(s)	Notes
A Arithmetic	Arithmetic, compare, **mfspr, mfcr0, mtcr0**	No special handling, these instructions can not cause exceptions. Single-cycle pipelined execution.
B Branch	**b, bc, bcctr, bclr**	All branches are fully resolved prior to fetching the next instruction – PPE 42 does not predict branch direction. Branches are decided in the **serial** state. When returning to **home** from **serial**, only taken branches update the IAR with the branch target.
C Context synchronizing	**mfmsr**	**mfmsr** is required to be execution synchronizing. In PPE it is context synchronizing as well. Because of the fact that parts of the MSR are updated by the memory interface, the memory interface must insure that the MSR is up-to-date before reporting completion of PIB data operations. 3 cycles + execution synchronization delay.
	mtmsr	**mtmsr** is required to be execution synchronizing. In PPE it is context synchronizing as well. **mtmsr** stalls for execution synchronization then does the normal writeback. 3 cycles + execution synchronization delay
	rfi	**rfi** is required to be context synchronizing. **rfi** is decoded as a branch to SRR0, and SRR1 ← MSR directly. 3 cycles + execution synchronization delay.
D Data	Load/Store/**dcb***	Potentially require blocking in the **data** state while waiting for memory. May cause DAC debug halts, Data Storage and Alignment Interrupts. May cause or report precise and imprecise data machine check interrupts. 2 cycle + memory-delay.
F Fused	Fused compare-branch	The fused compare-branch instructions are not pipelined. They execute serialized in 3 cycles. These branches are decided in the **fused** state.
K **stacK**	Load/Store Stack Frame (**lsku, lcxu, stsku, stcxu**)	The Stack Frame processing instructions added for PPE 42X are not pipelined. They execute serialized in multiple cycles as a programmable series of arithmetic operations and memory transactions. All notes from Class **D** also apply, except for the execution latency is between 4 and 14 cycles + memory-delays**.** Note that this is the only Class that can saturate the memory interface with back-to-back data transactions, which may occur during the **stack** state.
M Move To	**mtspr, wrtee, wrteei**	As a simplification all **mtspr** are handled specially and not pipelined. Many SPRs control exceptions, and we prefer a sequential execution model for instruction flow. The architecture does not require **wrtee** and **wrteei** to be context synchronizing but these may cause exceptions so they are handled here as well. Note that these instructions do not require synchronization. **mtspr**, **wrtee** and **wrteei** are simply delayed to allow any asynchronous exceptions that they may uncover to appear before fetching the next instruction. 2 cycles. **wrteei EE** decodes as **addi 0, 0, EE** with no GPR read or writeback. **wrtee Rn** decodes as **addi 0**, **RN**, **0** with no writeback. In both cases RDR[16] is copied directly to MSR[16] during **serialize**. Note that MSR[SIBRC] and MSR[SIBRCA] continue to update during execution of the **wrteei** and **wrtee** instructions.
S Sync	**sync**	**sync** stalls the pipeline for storage synchronization. For simplicity, **sync** is decoded as a **nop** and completes normally after synchronization. 3 cycles + storage synchronization delay.
T Trap	**tw (trap, mark)**	**mark** (**tw** 0,X,X) is executed as a single-cycle pipelined no-op. Other forms of **tw** (including **trap)** are never executed, but instead always causes an exception or

Instruction Class	Instruction(s)	Notes
		halt.

5 Exception Handling

Asynchronous and synchronous exception handling are disjoint, and unlike the Power ISA, all asynchronous exceptions have lower priority than synchronous exceptions.

Asynchronous exceptions are recognized only “between” instructions, technically before execution of the next instruction. PPE 42 never partially executes an instruction, and makes every attempt to complete every instruction fetched. If the state machine enters interrupt prioritization with an asynchronous interrupt pending, this can only be overridden by an imprecise machine check uncovered during execution synchronization.

Synchronous exceptions are recognized during instruction execution. If the state machine enters interrupt prioritization with a synchronous exception pending, there is no need to also check asynchronous exceptions, and the synchronous exception can only be overridden by an imprecise machine check uncovered during execution synchronization.

Asynchronous exceptions are only recognized during the **home** state. If an asynchronous event is pending and not overridden by a debug halt, then the fetch is aborted, and hidden state records the event for interrupt prioritization.

The **data** and **stack** state may cause data storage and precise machine checks, or allow imprecise data machine checks to be reported. The **sync** state will also report any pending imprecise data machine checks. Imprecise data machine checks are only reported during the **data**, **stack**, and **sync** states; the memory interface must hold imprecise machine checks until the next data access or synchronization event from the PPE 42. Note that by definition, all potential imprecise machine check sources must be reported before an execution synchronizing event (any interrupt or synchronizing instruction) is allowed to be executed.

Since the external interrupt input can glitch, it is captured as a hidden bit, but only in the **home** state. The timer interrupts (FIT, DEC and WDT) are captured in the **exception** state just before calculating priority, which occurs in the **vector** state. Therefore, external interrupts that happen to occur in the **sync, data, frame, stack, vector,** or **exception** state will not be seen until after the currently outstanding operation is completed, including taking any other interrupt that may be present while in those states, and the PPE 42 returns to the **home** state. At that point the external interrupt will be recognized if it still remains pending (was not a glitch) such that it can then participate in prioritization to be taken.

5.1.1 EDR and ISR handling

Note that in the event of an instruction or data machine check, the EDR and ISR status bits are updated simultaneously with the reporting of the event by the memory interface. This is valid because machine checks are always the highest priority interrupt outstanding, so these updates will never be incorrect, even though technically interrupt prioritization does not take place until after execution synchronization. If multiple data machine checks are presented the EDR and ISR may be updated again to reflect the latest machine check reported.

Data storage and alignment interrupts also immediately update EDR to avoid having to hold 32 bits of state for interrupt prioritization, overwriting the value being held in the EDR if in the **frame** or **stack** state. However if a data machine check is uncovered during prioritization, the EDR will be overwritten with the machine check address.

Program interrupts also immediately update EDR to avoid having to hold 32 bits of state for interrupt prioritization. However if a data machine check is uncovered during prioritization, the EDR will be overwritten with the machine check address.

The ISR status bits associated with instruction storage, data storage, alignment and program events are not updated until after interrupt prioritization however, so hidden state is required to hold this status.

6 Halting

The core is said to be in the halted state whenever

- State == **home**
- XSR[HCP] = '1'
- XSR[SIP] = '0'
- XSR[RIP] = '0'
- Hidden state bit “pipelined” is '0'.

Halting never synchronizes the processor core. An interrupt halt will synchronize the core as a natural part of interrupt processing, but the other halts stop the core immediately and present the core for debugging with the exact architected state that existed at the completion of the last completed instruction.

The core state machine only checks for a halted processor in the **home** state. If the processor is halted and we are not single stepping (XSR[SIP]] = '0') or ramming (XSR[RIP] = '0'), the fetch is aborted and the machine remains in the **home** state.

The single-step and ramming state is cleared before completion of the single-stepped or rammed instruction, so that when the state machine returns to the **home** state it will remain **home** (because XSR[HCP] = '1' when single stepping or ramming). Specifically, the XSR[SIP] bit is cleared when the fetch of the single-stepped instruction is ACKed by the I-side. The XSR[RIP] bit is cleared on the cycle that the instruction would normally update the IAR to IAR + 4. When ramming XSR[RIP] = '1' always suppresses this update

IAC and DAC debug halts are ignored when ramming.

The IAR reported in the halted state depends on why the processor halted:

- If the processor halts due to an IAC or DAC or **trap** debug event, the IAR addresses the instruction causing the event.
- If the processor is halted by external command, of if single-stepping, the IAR addresses the next sequential instruction.
- If the processor halts due to a machine check while MSR[ME] = '0', an unmaskable interrupt while MSR[UE] = '0', the IAR addresses the first instruction of the associated interrupt handler.

6.1.1 Force Halting

Force halting clears all hidden state and trumps even reset.

Force Halt conditions include:

- Watchdog Timer halt
- XCR write with CMD = Force Halt
- Hardware detected error

7 Resets

7.1 System Reset

System reset is treated like an interrupt that is taken immediately, regardless of the current state of the PPE 42 core. Conceptually there is an arc from every state (including the **reset** state itself!) to the **reset** state.

7.2 Reset Behavior

Reset is a special transaction type to the memory interface. Two types of reset are supported: A “soft” reset that instructs the memory interface to allow pending transactions to complete before ACKing, and a “hard” reset that forces an immediate reset.

System reset is implemented by a single state in the state machine; soft vs. hard is represented by a hidden state bit.

Reset behavior, as well as all sources of hard and soft reset, is fully specified in the PPE 42 User's Manual.

8 References

The *PPE 42 Core User's Manual* contains detailed user-level descriptions of the following behavior:

- Exception and interrupt handling
- Single-stepping and ramming procedures
- Behavior during ramming
- Synchronization
- Reset state changes and sequencing
- Stack frame processing instruction usage

To avoid possible inconsistencies if these descriptions were duplicated, those sections are referred from this document as the documentation of record.

9 State Machine Encoding

The only requirement is that state 0 is illegal.

The state encodes chosen for the original PPE42 and PPE42X implementation are as follows:

State	**Encode (hex)**	**Encode (binary)**	**Instruction Class**	**Summary Description**
home	1	0001	A, T	Execute instructions that can be pipelined (Class A, also Class T when traps are disabled). Complete all instruction types that execute without error. Process asynchronous interrupts. Also used for Wait state and Halt conditions.
data	2	0010	D, K	Perform memory transactions for Class D and K instructions.
serial	3	0011	B, C, F, K, M, S	Serialize non-arithmetic, non-data instructions that can not be pipelined. Commits ALU results for Class C, K, M, and S instructions. Computes branch address for Class F.
fused	4	0100	F	Computes branch address for fused compare-branch instructions.
sync	5	0101	C, K, S	Special synchronizing memory transactions. Performs “light-sync” for Class C & K instructions and “heavy-sync” for Class S.
exception	6	0110	-	Perform execution synchronization for interrupt handling (“light-sync” transaction)
vector	7	0111	-	Prioritize all pending interrupts, update the IAR with the interrupt vector, and update SRR0, SRR1, and MSR.
reset	8	1000	-	Handle two types of reset (soft and hard), which can occur from any state.
frame	9	1001	K	Calculate previous stack frame pointer & put in EDR for subsequent stack frame accesses.
stack	A	1010	K	Implement memory transactions for processing the stack frame.
<illegal>	0, B..F	-	-	Engine halts and sets internal error output.

Table 6: State Machine Encodings and Behavior Summary

10 State Machine

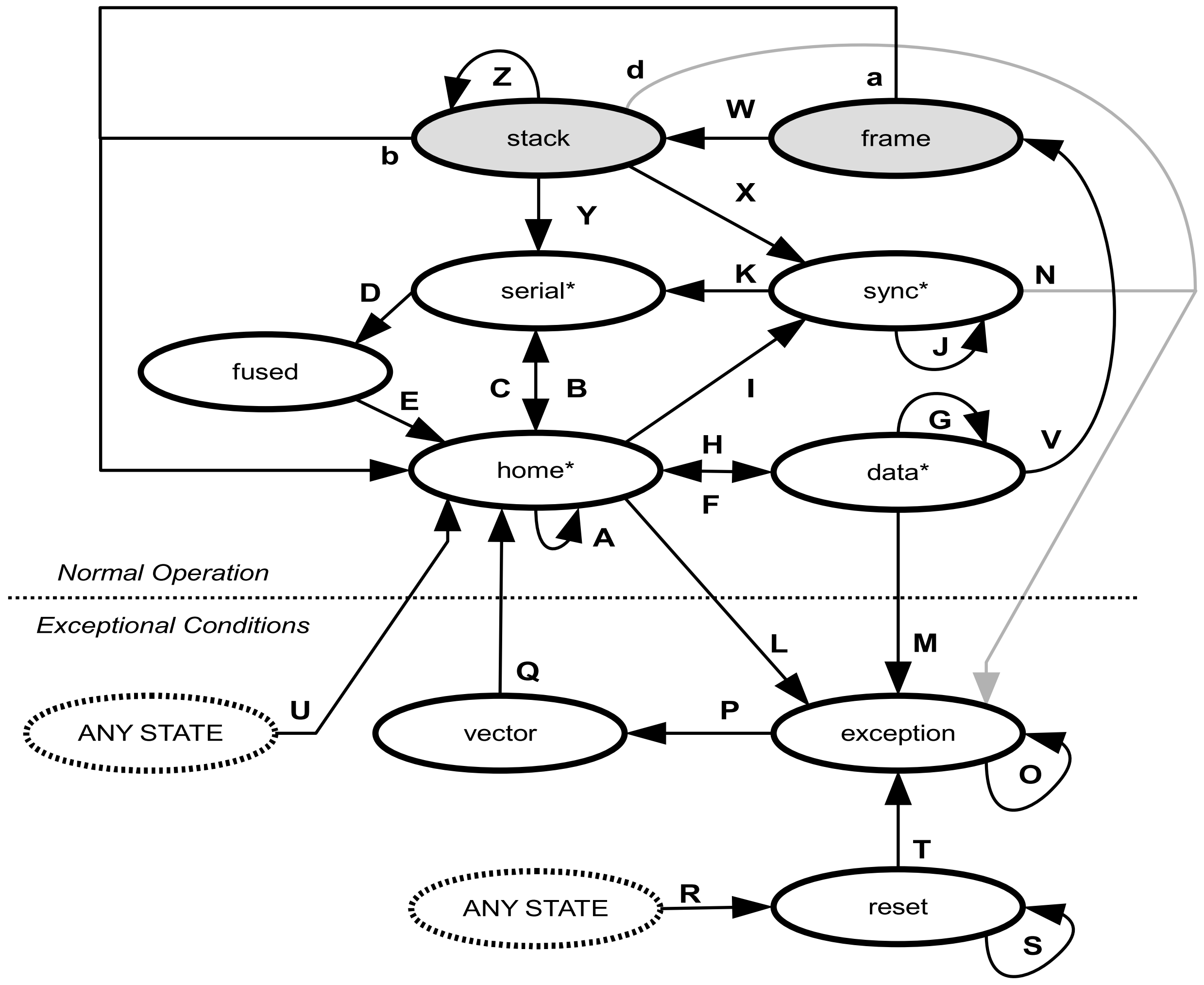


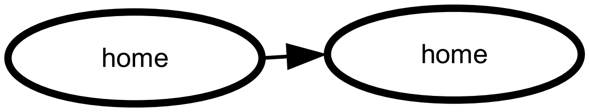
Illustration 2: PPE 42 & 42X State Machine. Note that the two shaded states (**stack** & **frame**) are only implemented by PPE 42X, and the four states with an asterisk* have some modifications for PPE 42X.

Table 7: PPE 42 & 42X State Transitions

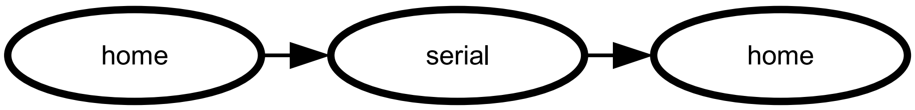
#	From	To	PPE42X Change	Notes
A	home	home		I-fetch nack, or ack a Class A instruction. Also spin here while halted or MSR[WE] = '1' Also return here for TRAP, IAC, DAC debug halts
B	home	serial		I-fetch ack Class B, F, or M instruction When ramming, Class C goes this way as well
C	serial	home	modify	Class B, K, M, C and S complete
D	serial	fused		Compute branch address for Class F
E	fused	home		Class F completes
F	home	data	modify	Fetch ack a Class D or Class K instruction
G	data	data	modify	Class D or K waits for memory
H	data	home		Class D completes
I	home	sync		Fetch ack Class C or S When ramming, only class S goes this way, Class C goes to **serial**
J	sync	sync	modify	Class C, K, or S waits for synchronization
K	sync	serial	modify	Go complete Classes C, K, and S
L	home	exception		Fetch or decode phase exception, including non-halting **trap**
M	data	exception	modify	Memory exception or Class K alignment exception
N	sync	exception		Imprecise data machine check uncovered by synchronization
O	exception	exception		Waiting for synchronization
P	exception	vector		Sync done; Prioritize interrupts and compute vector
Q	vector	home		Resume after exception
R	ANY State	reset		Reset. This arc is trumped by a force halt.
S	reset	reset		Wait for reset ack from MIB
T	reset	exception		Take the SRESET exception
U	ANY State	Home		Force halt. This arc trumps arc R (reset).
V	data	frame	new	Class K updates previous stack frame link
W	frame	stack	new	Class K start processing stack frame content
X	stack	sync	new	Class K requires synchronization when stku in imprecise mode (MSR[IPE]=1)
Y	stack	serial	new	Go complete Class K when lsku or if in precise mode (MSR[IPE] = 0)
Z	stack	stack	new	Class K waits for memory
a	frame	home	new	Class K takes a debug halt before accessing stack frame content
b	stack	home	new	Class K takes a debug halt when accessing stack frame content
d	stack	exception	new	Memory exception when accessing stack frame content

State traversals by instruction class for PPE42, assuming a fetch-acknowledgment in the first home state

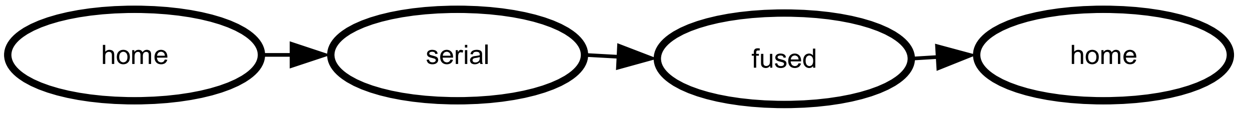
Class A; Also Class T when DBCR[TRAP] = '1'



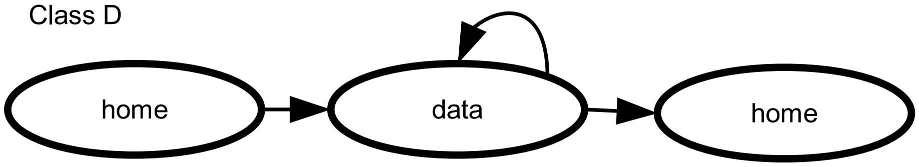
Classes B, M; Also Class C when ramming



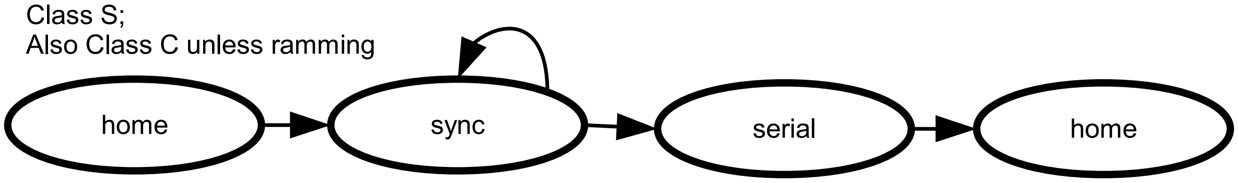
Class F



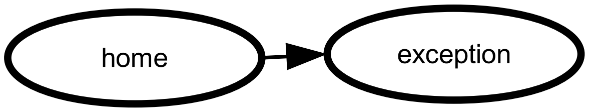
Class D



Class S;
Also Class C unless ramming



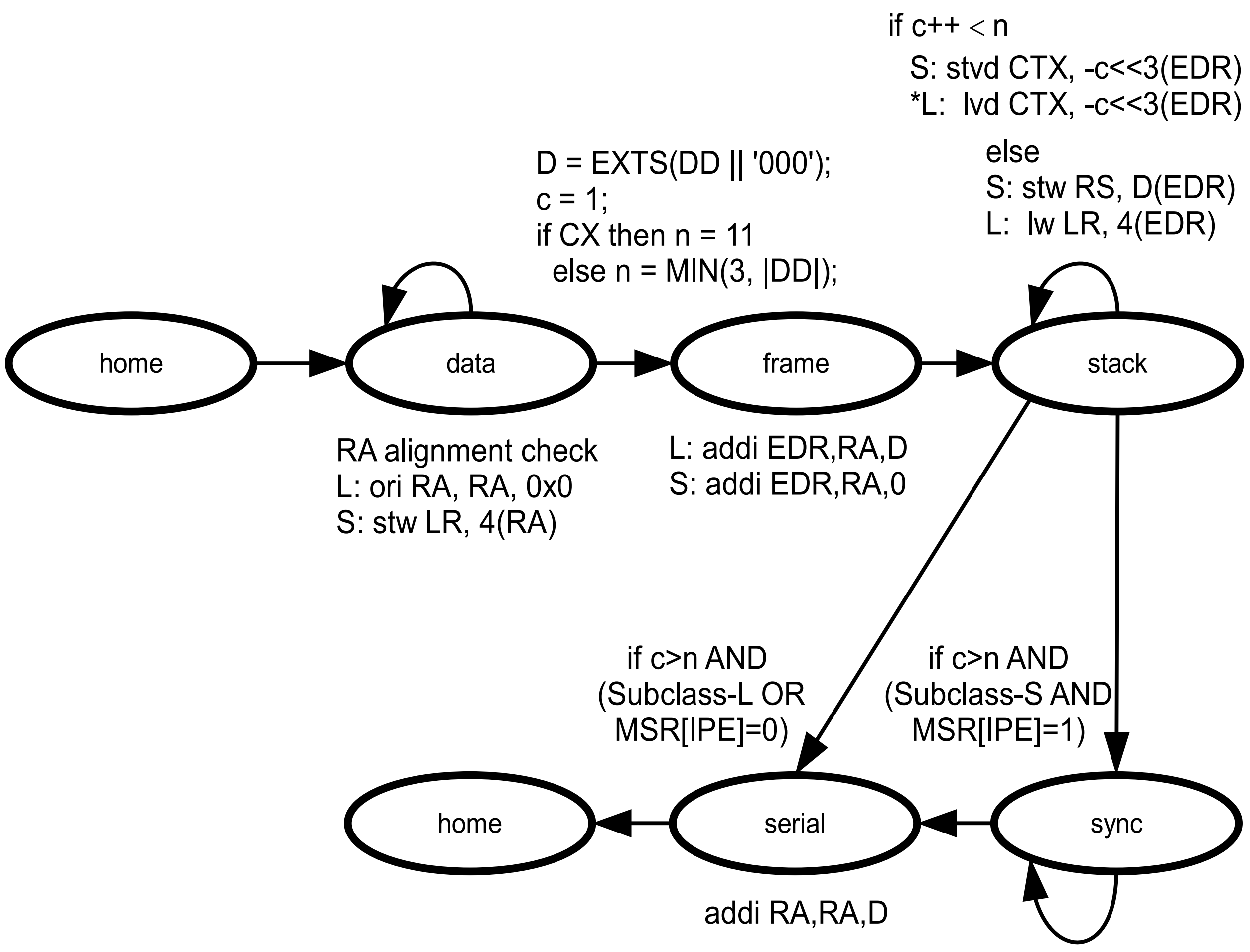
Class T when DBCR[TRAP] = '0'



New State traversals for instruction class K for PPE42X, assuming a fetch-acknowledgment in the first home state

Class K: lcxu, lsku, stcxu, stsku

Sub-class of L: (for lsku/lcxu) and S: (for stsku/stcxu) delineate the different operations done for each.
Sub-class of CX for lcxu and stcxu
note: sync state is bypassed for subclass L, and in Precise Mode for subclass S



* Note for lcxu and lsku: for the lvd in the stack state, updates to RA are suppressed.
i.e. if the VDR contains RA, only one of the two doublewords are written into the register file.
For example, if RA==1; the “lvd D0” would update R0 but not R1.
Likewise, if RA==3, the “lvd D3” would update R4 but not R3.

11 State Tables

The tables in this section detail the low-level operations of the core. Conditions and state transitions are prioritized, and must be processed in priority order. Mutually exclusive conditions are always considered at the same priority.

11.1 home

At entry to the **home** state a hidden state bit may indicate that a pipelined Class A instruction was decoded on the previous cycle. If so, the pipelined instruction is always completed concurrent with the fetch or other state transition.

The **home** state effectively has 3 “substates”:

- **default** : Normal execution
- **step** : Single stepping, i.e., XSR[SIP] = '1'. XSR[SIP] is always cleared once the single-step fetch completes or takes an error.
- **ram** : XSR[RIP] is set when the IR is written. XSR[RIP] is cleared here for instructions that complete without exception, however for other instructions XSR[RIP] remains set to squash the subsequent IAR update, and is cleared when the instructions complete.

Ramming ignores most conditions and simply decodes and executes the instruction held in the IR.

The IAR is only updated for **default** and **step** fetches when the instruction will complete without an exception. For Class D, K, and S instructions we wait until they complete without exception before updating the IAR. This eliminates the need for IAR rollback.

For Class K instructions, the instruction decode to the ALU, GPR selects, and SPR selects is overridden as if a different instruction was decoded to perform the first step of the stack frame processing.

There is no “wait” state. If MSR[WE] = '1' then under normal conditions we spin here waiting for something that allows the state machine to continue. If single stepping we are a hermit and never leave home.

Table 8: The **home** state

Priority	Substates	Condition	Squash Fetch?	Next State	Updates / Special	Notes
ANY					Any pipelined instruction is completed.	

1	**ram**	Illegal decode, or **trap** with DBCR[TRAP] = '0'	Yes	**exception**	Exception hidden state;
					EDR ← instruction;	

	ram	**trap** with DBCR[TRAP] = '1'	Yes	**home**	XSR[HCP] ← '1' XSR[HC] ← '100' XSR[TRAP] ← '1'	
	ram	IR decodes as Class A	Yes	**home**	Set “pipelined” hidden state; XSR[RIP] ← '0'	
	ram	IR decodes as Class B, C, F, M	Yes	**serial**	XSR[RIP] ← '0'	
	ram	IR decodes as Class K-L	Yes	**data**	XSR[RIP] unchanged	Class K-L sets up a ori RA, RA,0 decode
	ram	IR decodes as Class D or K-S	Yes	**data**	XSR[RIP] unchanged	If Class K-S, sets up a decode of: stw LR, 4(RA)
	ram	IR decodes as Class S	Yes	**sync**	XSR[RIP] unchanged	
2	**default**	XCR[HR] = '1'	Yes	**home**		Halted and not single-stepping
3	**default** **step**	Asynchronous exception pending	Yes	**exception**	Set asynchronous exception hidden state	All of these transitions clear XSR[SIP]
4	**default** **step**	MSR[WE] = '1'	Yes	**home**		
5	**default** **step**	IAC debug Halt	Yes	**home**	XSR[IAC] ← '1' XSR[HCP] ← '1' XSR[HC] ← '100'	
6	**default** **step**	I-fetch NACK	No	**home**		
	default **step**	I-fetch error	No	**exception**	Set hidden state bits for exception; ISR updates for machine check.	All of these transitions clear XSR[SIP]
	default **step**	Illegal decode, or **trap** with DBCR[TRAP] = '0'	No	**exception**	Set hidden state bits for exception; EDR ← instruction	
	default **step**	**trap** with DBCR[TRAP] = '1'	No	**home**	XSR[HCP] ← '1' XSR[HC] ← '100' XSR[TRAP] ← '1'	
	default **step**	I-fetch ACK decodes as Class A or Class T non-**trap** (either **mark** or **marktag**)	No	**home**	IAR += 4; Set “pipelined” hidden state bit	
	default **step**	I-fetch ACK decodes as Class B, F, M	No	**serial**	IAR += 4	
	default **step**	I-fetch ACK decodes as Class K-L	No	**data**	IAR unchanged	Class K-L sets up a ori RA, RA,0 decode
	default **step**	I-fetch ACK decodes as Class D or Class K-S	No	**data**	IAR unchanged	If Class K-S, sets up a decode of: stw LR, 4(RA)
	default **step**	I-fetch ACK decodes as Class C, S	No	**sync**	IAR unchanged	

11.2 data

The **data** state implements memory transactions.

At entry the ALU has been programmed to compute the data address.

The **data** state effectively has 4 “substates”:

- **default** : Normal execution for Class D
- **ks** : Normal execution for Class K-S (stsku or stcxu)
- **ls** : Normal execution for Class K-L (lsku or lcxu)
- **ram** : XSR[RIP] is set when the IR is written. Note that XSR[RIP] is cleared here when the instruction completes, if not Class K.

For Class K-S, the Link Register is stored to a word in memory at the calculated data address if RA is doubleword aligned, meaning the output of ALU (RA+4) is odd word aligned (i.e. last three bits = “100”).

For Class K-L, the ALU performs a no-op update to RA so output of ALU can be checked for doubleword alignment (i.e. last three bits = “000”).

For Class D, the IAR is updated here, but only if the instruction completes without an exception. This eliminates the need for IAR rollback.

If not ramming, Debug Address Compare is enabled in DBCR to cause a debug halt, and an address match to DACR occurs, any GPR update is prevented and the memory transaction is squashed (i.e. not presented to the memory interface).

Table 9: The **data** state

Priority	Substates	Condition	Squash Data Transaction?	Next State	Updates / Special	Notes
1	**default**	DAC debug halt (ALU output matches DACR)	Yes	**home**	XSR[HCP] = '1' XSR[HC] ← '100' XSR[DACR, DACW] updates Clear XSR[RIP, SIP]	GPR update and Data Transaction are both squashed.
2	**default** **ks** **ram**	D-side NACK	No	**data**		Wait for memory interface
	default **ks** **ram**	D-side error	No	**exception**	Set exception hidden state; ISR and EDR updates for machine checks; EDR update for data storage and alignment.	Clear XSR[RIP, SIP]
	kl **ram**	If Class K-L and ALU output (RA) is not doubleword aligned	Yes	**exception**		
	ks **ram**	If Class K-S and ALU output (RA+4) is not odd word aligned				
	default **ram**	D-side ACK if not Class K	No	**home**	Commit store data and/or GPR update to GPRs; If ~XSR[RIP] then IAR += 4 XSR[RIP] ← '0'	
	ks **ram**	D-side ACK if Class K-S	No	**frame**	Commit store data from LR	Sets up a decode of: addi EDR, RA, 0
	kl **ram**	Class K-L	No	**frame**	No memory transaction.	Sets up a decode of: addi EDR, RA, D

11.3 serial

The **serial** state is used to serialize non-arithmetic, non-data instructions that can not be pipelined. XSR[RIP, SIP] are cleared here for instructions that complete here.

For Class K instructions, the serial state is used to perform the update of the stack pointer in the base address register.

Table 10: The **serial** state

Priority	Condition	Next State	Updates / Special	Notes
1	Class F instruction	**fused**	Update CR[CR0] from ALU. Set ALU decode to compute the fused compare-branch target.	All of these transitions clear XSR[RIP, SIP]
	Class B instruction	**home**	Update IAR from ALU if taken	
	Class K instruction	**home**	IAR += 4 Commit RA update from ALU	
	Class M, S instructions	**home**	Commit GPR/SPR updates from the ALU	
	Class C other than **rfi**	**home**	Commit GPR/SPR updates from the ALU	
	rfi	**home**	Update IAR from ALU; MSR ← SRR1	**rfi** moves SRR0 through the ALU. Clear XSR[RIP, SIP]

11.4 fused

The **fused** state exists to implement the branch address computation for fused compare-branch instructions.

Table 11: The **fused** state

Priority	Condition	Next State	Updates / Special	Notes
1	All	**home**	Update IAR if branch taken	Clear XSR[RIP, SIP]

11.5 sync

The **sync** state implements the special synchronizing memory transactions. The **sync** state does a "heavy-sync" for Class S and a "light-sync" for Class C and Class K.

Table 12: The **sync** state

Priority	Condition	Next State	Updates / Special	Notes
1	D-side NACK	**sync**		
	D-side error	**exception**	Set exception hidden state; ISR and EDR updates for machine checks.	Clear XSR[RIP, SIP]
	D-side ACK for Class S or C	**serial**		
	D-side ACK for Class K	**serial**		Sets up a decode of: addi RA, RA, D

11.6 exception

The **exception** state exists to perform execution synchronization for interrupt handling. The **exception** state does a "light-sync" transaction.

Table 13: The **exception** state

Priority	Condition	Next State	Updates / Special	Notes
1	D-side NACK	**exception**		
	D-side error	**vector**	Set exception hidden state; ISR and EDR updates for machine checks	
	D-side ACK	**vector**		

11.7 vector

The **vector** state exists to prioritize all pending interrupts and update the IAR with the interrupt vector. SRR0 gets IAR directly, and the MSR updates for the taken interrupt are also programmed here. The table below discusses conditions *after* interrupt prioritization

Table 14: The **exception** state

Priority	Condition	Next State	Special	Updates

1	Machine check interrupt and MSR[ME] = '0', or other unmaskable interrupt and MSR[UE] = '0'.	**home**	XSR[HCP] ← '1'
			XSR[HC] ← '011'
				Any ISR updates from hidden state.
				All interrupt hidden state cleared.
				SRR0 ← IAR
				IAR ← IVPR || <vector>.
				MSR cleared except for MSR[ME], machine check also clears ME.

	Otherwise	**home**		

11.8 reset

There is an implicit transition to reset from every state. There are 2 types of reset, soft and hard, which are presented externally as special memory transactions.

The following reset actions must occur on the transition into the **reset** state from any other non-**reset** state:

- ISR[SRSMS] is updated from the current state machine state.

For further information on the **reset** state and the data updates see the PPE 42 User's Manual.

Table 15: The **reset** state

Priority	Condition	Next State	Updates
1	Reset NACK	**reset**	
	Reset ACK	**exception**	Set reset exception hidden state bit

11.9 frame

The **frame** state is used by Class K instructions to calculate the previous stack frame pointer and place it in EDR for subsequent stack frame data accesses. Substate **kl** is defined for Subclass K-L (***lsku*** and ***lcxu***). Substate **ks** is defined for Subclass K-S (***stsku*** and ***stcxu*****).**

For **kl** only, Debug Address Compare matches, when enabled, cause a halt to prevent GPR or LR state from being subsequently modified. The compare is not done for **ks** since EDR is being written with address of the previous stack pointer, which does not correspond to the base address plus displacement specified by the instruction.

The number of stack frame transactions is one if the instruction fields DD=1 or DD= -1. In this case, the only thing to do is set up decode to restore the Link register for **kl** and update the back pointer for **sk**. Otherwise, set up decode to perform the first doubleword operation.

Table 16: The **frame** state

Priority	**Substate**	**Condition**	**Next State**	**Updates / Special**	**Notes**
1	**kl**	DAC debug halt	**home**	XSR[HCP] = '1' XSR[HC] ← '100' XSR[DACR, DACW] updates Clear XSR[RIP, SIP] EDR is updated (not squashed)	Compares output of the ALU with the DACR (like **data**). Does not compare for substate **ks.**
2	**kl** **ks**	DD=1 or DD=-1	**stack**	Prepare to process the stack frame: • EDR is updated with address of the previous stack pointer from ALU • The hidden state count c is set to 1	Sets up a decode of: lw LR, 4(EDR)
					Sets up a decode of: stw RS, D(EDR)
	klks	DD != 1 and DD != 1	**stack**		Sets up a decode of: stvd D30, -8(EDR)
					Sets up a decode of: lvd D30, -8(EDR)

11.10 stack

The **stack** state is a more complex version of the **data** state that implements the memory transactions for Class K instructions to process the stack frame. At entry the ALU has been programmed to compute the first data address to access. The ALU is used to compute the next data address every time a D-side ACK is received until the selected stack frame data has been completely processed.

The **stack** state calculates the number of transactions *n* to the stack frame. For ***lcxu*** and ***stcxu*** instructions, this number *n* is 11, else for ***lsku*** and ***stsku*** the number *n* is determined by either the magnitude (absolute value) of the DD field, or the number 3, whichever is smaller. This means *n* always equals 1,2,3, or 11.

CTX[c] is a list of doublewords from [1..10] comprising context that is placed on the stack frame, defined from a subset of GPRs and SPRs in this order:
VDR(30), VDR(28), (SRR0 || SRR1), (XER || CTR), VDR registers (9,7,5,3,0), and (CR || SPRG0). Note that if RA is contained in the VDR, the doubleword is fetched from memory but update of RA is suppressed.

The count of memory transactions performed is incremented by D-side ACKs has already been set to 1 upon entry to this state to reflect the transaction that is initiated on the first cycle of the **stack** state.

Table 17: The **stack** state

Priority	Substates	Condition	Next State	Updates / Special	Notes
1	**kl** **ks**	DAC debug halt	**home**	XSR[HCP] = '1' XSR[HC] ← '100' XSR[DACR, DACW] updates Clear XSR[RIP, SIP] Clear hidden state count c = 0	Data Transaction is squashed (not presented to the memory interface)
2	**kl** **ks** **ram**	D-side NACK	**stack**		Wait for memory interface
	kl **ks** **ram**	D-side error	**exception**	Set exception hidden state; ISR and EDR updates for machine checks; EDR update for data storage and alignment. Clear hidden state count c = 0	
	lk **ram**	c = n and D-side ACK	**stack**	Loads update the selected VDR from memory, except that update of RA is suppressed. Stores update memory with the specified VDR. Initiates the next doubleword stack frame transaction to the memory interface; hidden state count c is incremented.	Sets up a decode of: lw LR, 4(EDR)
	sk **ram**		**stack**		Sets up a decode of: stw RS, D(EDR)
	lk **ram**	c < n and D-side ACK	**stack**		Sets up a decode of: stvd CTX[c], -8(EDR)
	sk **ram**		**stack**		Sets up a decode of: lvd CTX[c], -8(EDR)
	sk **ram**	c > n and D-side ACK when MSR[IPE]=1	**sync**	Final store of back pointer to stack frame header. Clear hidden state count c = 0	
	sk **ram**	c > n and D-side ACK when MSR[IPE]=0	**serial**		Sets up a decode of: addi RA, RA, D
	lk **ram**	c > n and D-side ACK	**serial**	Final load updates LR from stack. Clear hidden state count c = 0	