The AJIT processor

Madhav P. Desai
Department of Electrical Engineering, IIT Bombay

September 27, 2023

Contents

Acknowledgements

Several persons have contributed to the technology described in this user guide. I would specifically mention Neha Karanjkar for her contributions to the C reference model, peripheral device integration and contributions to the memory subsystem modeling, the compilation tool setup and Linux support. Titto Thomas made important contributions to the GDB remote debugger support. Others who contributed are Piyush Soni (for the verification setup of the processor), R. Aswin (for floating point unit design), M. Sharath (for the first C model of the processor). The initial funding for the development of the processor was provided by MEITY (Govt. of India) and Powai Lab Technologies Pvt. ltd.

Chapter 1

AJIT Processor Family

The AJIT processor family consists of a variety of configurations and is intended for high performance embedded system and IOT applications. The processor core configurations range from single-core single-thread implementations of the SPARC-V8 instruction set architecture to quad-core octa-thread implementations of the SPARC-V8-AJIT64 instruction set architecture.

The structure of the AJIT processor family is shown in Figure 1.1. The family is rooted in the two CPU implementations. CPU32 is an implementation of the SPARC-V8 ISA, and CPU64 is an implementation of the SPARC-V8 + AJIT-64 ISA, which includes 64-bit and SIMD ALU, FP instructions. A core is constructed by putting together a CPU with the memory unit MUNIT, which is assembled using an ICACHE, DCACHE and MMU.

1.1 AJIT core building blocks

The processors are built around the following building blocks.

- A central processing unit (CPU32), also known as a **thread**, which implements the SPARC-V8 (IEEE 1754-1994) instruction set architecture. An extended version of the CPU (CPU64) is also implemented. The AJIT CPU64 supports the SPARC-V8 ISA, and also supports vector instructions for DSP/AIML/Embedded applications.

- A memory unit (MUNIT) which includes data and instruction caches, as well as a memory management unit.

- A coherent memory controller with snoop based cache coherence.

- A programmable interrupt controller with a per thread interrupt state machine.

- Each AJIT processor can be configured with up to four AJIT cores.

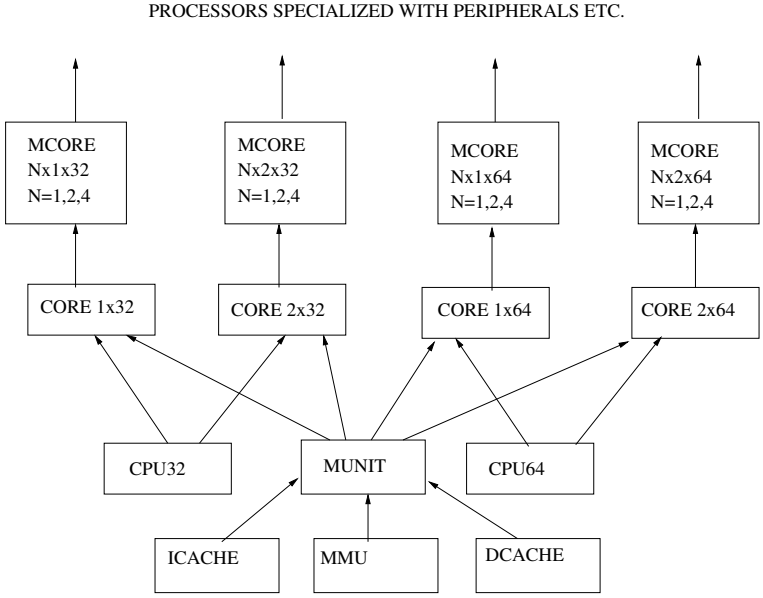

Figure 1.1: AJIT processor family structure

- Each AJIT core can be configured with either one or two hardware threads (CPU's).
- Each AJIT CPU can have two flavours:
  - The base CPU is an implementation of the IEEE std. 1754-1994 (SPARC V8) instruction set. We call this the AJIT CPU32.
  - An extended version of this CPU implements the SPARC V8 instruction set with extensions that support 64-bit integer operations as well as vector operations on 8/16/32-bit integer data. We call a CPU with this extensions as the AJIT CPU64.

Thus, an AJIT processor can have a minimum configuration of one core, one cpu thread per core, with each CPU thread being an AJIT CPU32 (implementing the base SPARC-V8 ISA). The maximum configuration of an AJIT processor consists of four cores, each of which has two CPU threads, with each CPU thread being an AJIT CPU64 (implementing the SPARC-V8 ISA plus vector extensions), and is shown in Figure 1.2. This provides a wide array of choices to an embedded system designer.

1.2 Memory subsystems in AJIT processor cores

Each AJIT core has a full featured memory subsystem. This memory subsystem is shared by the CPU threads within an AJIT core.

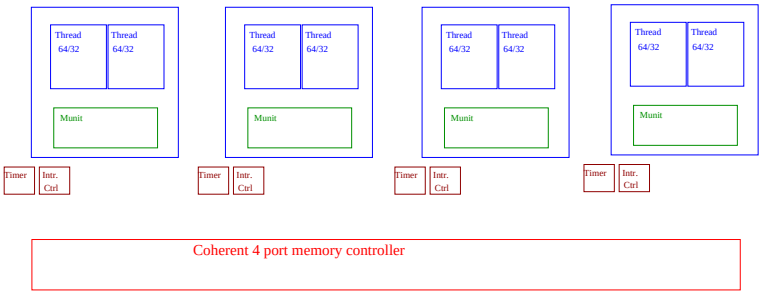

Figure 1.2: AJIT processor with 4 cores, each with 2 threads

- The AJIT memory unit (MUNIT) includes 4 KB to 32 KB set associative (1-way 8-way) virtually indexed, virtually tagged (VIVT) instruction cache (ICACHE), a VIVT configurable 4KB to 32 KB set associative (1-way to 8-way), write-through-allocate data cache (DCACHE). The cache access latency on a hit is 1 clock cycle. The processor core also includes a memory-management-unit (MMU), an implementation of the Sparc-reference MMU (SRMMU) with an 256-entry level 3 page translation-lookaside-buffer (TLB), a 16-entry level 2 page TLB and a 4-entry level 1 page TLB. On TLB misses, the MMU is capable of page translation autonomously, using a 4-level page table walk.

- The memory subsystem includes a synonym detection and avoidance mechanism to prevent multiple copies of the same memory data from being present in the caches.

- The memory subsystem includes a snoop based cache coherence mechanism for building coherent shared memory multi-core systems.

1.3 Debug support mechanisms in AJIT processor cores

Each AJIT processor core has sophisticated debug support mechanisms for comprehensive debugging of system code.

- For debug support, each AJIT CPU thread includes a dedicated hardware debug unit which offers complete observability and controllability of internal processor state. The hardware debug unit can be used in conjunction with the GDB debugger to offer real-time debug capability for the AJIT CPU thread.

- Each AJIT CPU thread can also generate an execution trace which can be captured (at run time) and stored into a trace buffer which uses the system memory.

1.4 Micro-architectural details about the AJIT CPU threads

Each AJIT CPU thread has a single issue, in-order pipeline. Thus, the CPU thread is not vulnerable to micro-architectural side-channel attacks. The pipeline runs at a rate of 1 instruction per cycle, and is a 7 stage pipeline with full floating point (IEEE 754) support. The pipeline includes an integer divider, as well as floating point square-root and divider (single and double precision) units.

The CPU includes branch prediction and speculative execution of streams. It has a 256-entry branch translation buffer, and uses a 2-bit history to predict branch decisions. It also includes a 16-entry return address stack for predicting subroutine returns.

The CPU has a fully pipelined load/store unit with total load/store ordering, and a pipelined floating point unit (single precision and double precision). Denormalized numbers as specified in IEEE 754 are fully supported. Floating point operations (all, except divide and square root) are pipelined, and normally take two clock cycles. If the result of an operation needs to be delivered as a denormal (a rare event), then an additional two cycles are consumed.

1.5 AJIT CPU: exceptions and interrupts

An AJIT CPU has a well defined set of exceptions (as defined in the SPARC-V8 standard), which includes hardware as well as software generated exceptions. Further, each AJIT CPU can respond to 15 possible interrupt levels. Trap handling is done transparently by the CPU using a vectored trap mechanism. Interrupts are sampled at every instruction, in the decode stage of the pipeline.

All exceptions are handled **precisely** (including floating point exceptions). This simplifies the job of writing exception handlers since the state of the processor at the point of appearance of the exception is preserved exactly. The minimum trap and interrupt latency of the AJIT processor core is 12 clock cycles.

1.6 AJIT processor interfaces

The AJIT processor core has the following interfaces to the outside world:

- A 64-bit system bus interface called the AJIT-CORE-BUS (ACB), which consists of a request-response pair of FIFO interfaces. Bridges to AXI and AHB interfaces are available to interface to a wide variety of memory and peripheral devices. The ACB is described in detail in Chapter 7.

- A 4-bit interrupt level input: the external world can signal an interrupt with up to 15 interrupt classes to the CPU. Interrupt level 15 is non-maskable.

- A debug interface: this is a request/command interface which is used by external remote debug tools (e.g. GDB) for in-system debugging. The debug interface can be equipped with a serial UART or JTAG.
- Reset control and mode status: A 4-bit reset control signal is provided to control the reset sequence of the AJIT processor. A 2-bit CPU mode signal is generated by the CPU to indicate its current mode of operation.

These interfaces are described in more detail in Chapter 2.

1.7 AJIT development tools

A complete software development chain is available. The existing GCC compiler, GDB debugger and the Linux kernel for the SPARC-V8 ISA are compatible with the AJIT processor core. Binary utility support for both AJIT32 and AJIT64 CPU's is available.

In addition,

- A full featured processor emulator (written in C) is available for use by software developers and system designers.
- FPGA prototypes of the processor configurations listed earlier are also available.
- A suite of tools for software development (trap handlers, timer control, interrupt handling, printing etc.) and compilation scripts are available for systematic management of the entire software development process.
- A cooperative real-time operating system (CORTOS2) is available for application development. This comes with a build system which simplifies the mapping of an application to a multi-core, multi-thread AJIT processor core.

1.8 AJIT Processor core: ASIC history

A proof-of-concept implementation in 180nm technology was carried out at SCL Chandigarh.

- 7mm X 7mm, 144 functional I/O's, 256 pin package.
- 0.5 W at 66 MHz (slow corner).

An implementation of the AJIT processor core for a NAVIC GPS+IRNSS receiver system-on-chip has been fabricated in 65nm technology at UMC (Taiwan). The SOC is completely functional based on first silicon testing.

- The total area of the SOC is 4mm X 4mm (56 I/O, 100 total pads). The are occupied by the CPU is 1mm X 1mm.

- The CPU can run at 500MHz, but the overall system is operated at 125 MHz with 512KB of single-cycle SRAM being in the critical path.
- System power dissipation is 250mW @ 125MHz. Out of this, the processor power dissipation is 50mW.

1.9 Looking ahead

In the rest of this document, we will describe:

- Multi-core features.
- Single-core features.
- The AJIT CPU in two flavours: base SPARC-V8 (AJIT CPU32) and base SPARC-V8 with 64-bit and vector extensions (AJIT CPU64).
  - Exceptions and trap numbering.
- The AJIT core memory unit.
  - Data and Instruction caches, Memory-management-unit.
- Peripherals currently available for the AJIT core.
- Tools: for building applications.
- Available FPGA prototypes.
- Debug infrastructure using on-CPU hardware debug unit, as well as GDB remote debug process.

Chapter 2

The AJIT CPU32 execution thread

We describe the central processing unit (CPU) of the AJIT processor core. Often, we will refer to the CPU as a *thread*.

2.1 Instruction Set Architecture

An AJIT thread implements the SPARC-V8 instruction set architecture (draft IEEE standard 1754-1994) [1]. Specific decisions made in the implementation are listed below. All design decisions are consistent with the SPARC V8 ISA specification.

2.1.1 Instruction-wise implementation decisions

The instructions in the SPARC V8 ISA fall into the following classes: data-transfer instructions (listed in Figure 2.3), miscellaneous instructions (listed in Figure 2.4), integer ALU instructions (listed in Figure 2.5), integer TICC instructions (listed in Figure 2.6), control transfer instructions (listed in Figure 2.7), and floating point instructions (listed in Figure 2.8).

- **Data-transfer instructions**: All data transfer instructions are implemented, except for the following (which are optional in the SPARC-V8 ISA).

```
STDFQ     Not implemented (no FP queue)
STC       Not implemented (no Co-processor)
STDC      Not implemented (no Co-processor)
STCSR     Not implemented (no Co-processor)
STDCQ     Not implemented (no Co-processor)
```

- **Miscellaneous instructions**: All miscellaneous instructions are implemented.

- **Integer ALU instructions**: All integer ALU instructions are implemented.

- **Floating-point instructions**: All required floating point instructions are implemented. The following Quad-precision floating point instructions are not implemented.

```
FqTOi      Not implemented
FsTOq      Not implemented
FdTOq      Not implemented
FqTOs      Not implemented
FqTOd      Not implemented
FSQRTq     Not implemented
FADDq      Not implemented
FSUBq      Not implemented
FMULq      Not implemented
FdMULq     Not implemented
FDIVq      Not implemented
FCMPq      Not implemented
FCMPEq     Not implemented
```

- **Control transfer instructions**: All required control transfer instructions are implemented. Since there is no coprocessor included in the CPU, coprocessor branches are not implemented.

- **Trap instructions**: All trap instructions are implemented.

2.1.2 Exceptions

In Figure 2.1, we list the exceptions are generated and handled by the AJIT processor core. All exceptions are handled in a **precise** manner. If there is an exception, the processor generates a 32-bit trap-vector (TV). We list the indices of the trap-vector which correspond to the distinct exceptions.

On a trap, the CPU jumps to a vectored trap location. This location is determined as

```
(TBR & (~TT_MASK)) | (TRAP_ID << 4)
```

where TBR is the trap base register, $TT\_MASK = 0xffffff00f$, and TRAP_ID is the trap identifier. For the traps listed above, the identifiers are shown in Figure 2.2.

Mnemonic	TV-index	explanation
RT	0	reset-trap
IAE	4	instruction-access-exception
AT	3	annul-trap
PI	8	privileged instruction.
II	9	illegal instruction
FPD	11	fp-disabled
CPD	12	co-processor disabled
WOF	14	window-overflow
WUF	15	window-underflow
UA	16	unaligned-address
FPE	17	fp-exception invalid-reg sequence-error fp-unimpl-instr fp-ieee-754-trap
DAE	20	data-access-exception.
TOF	21	tag-overflow
IDZ	22	iu-div-by-zero
TT	23	trap-instruction-trap

Figure 2.1: Exceptions generated by the AJIT processor core

```
  ------------------------------------------------------------
  Mnemonic  TRAP_ID
  ------------------------------------------------------------
  IAE       0x1
  II        0x2
  PI        0x3
  FPD       0x4
  WOF       0x5
  WUF       0x6
  UA        0x7
  FPE       0x8
  DAE       0x9
  TOF       0xA
  CPD       0x24
  IDZ       0x2A
  TT        0x80 | ticc_trap_type

interrupt (when interrupt-level != 0)
  INTR      0x10 | interrupt-level
  ------------------------------------------------------------
```

Figure 2.2: Trap Identifiers

2.2 ASR mappings

The following ancillary state registers (ASR's) are used by the AJIT CPU implementation.

- **Free running counter**: ASR[31] and ASR[30] implement a free running 64-bit counter which runs at the processor clock. Writes to these registers are ignored.
- **CPU Identity**: ASR[29] stores a 32-bit value (read-only) which has the following format

```
[31:16]   RESERVED
[15:8]    Core Id to which CPU
              belongs
[7:0]     CPU Id within the core.
```

All other ASR's are implemented as R/W registers without any assigned mappings.

2.2.1 Listing of instructions

We list the instructions in the SPARC-V8 ISA and their status within the AJIT CPU implementation. For each instruction, we also indicate the exceptions that

it can raise.

In Figure 2.3, we show the data-transfer instructions.

In Figure 2.4, we show the miscellaneous instructions.

In Figure 2.5, we show the integer unit instructions.

In Figure 2.6, we show the trap (TICC) instructions.

In Figure 2.7, we show the control transfer instructions. Note that co-processor branch instructions are not currently implemented.

In Figure 2.8, we show the floating point instructions.

The co-processor operations are not implemented (Figure 2.9).

2.2.2 AJIT CPU (Thread) internal structure

The internal structure of the AJIT CPU (or thread) is shown in Figure 2.10.

The CPU contains

- A single issue 7-stage (instruction-fetch, decode, operand-fetch, execute, load-store, write-back, retire) execution pipeline, with a 256 entry branch history table and a 1-bit branch predictor.
- An IEEE 754 compliant floating point unit with full support for denormalized numbers, and built-in hardware divide and square-root units.
- A hardware debug support unit which allows full observability and controllability of processor state. It also allows remote real-time debugging via GDB.
- Precise exceptions, low interrupt latency (minimum of 12 cycles).

The caches are virtually indexed, virtually tagged and can be accessed in two clock cycles (on a hit). The Cache line size is 64 bytes, and the miss penalty is 40 cycles. The MMU implements the SPARC SRMMU standard, with a 4-level page table supporting pages of size 4KB, 256KB, 16MB and 4GB.

2.2.3 Instruction timings in the AJIT CPU

Most instructions in the AJIT CPU take one clock cycle to execute. The time taken by control transfer instructions and load/store instructions is non-deterministic.

- Most integer operations are single cycle, except for divides.

```
Operation  Cycles Pipelined?
Logical       1      Yes
Arithmetic    1      Yes
Divide       21      No
```

- Most floating point operations are pipelined with a latency of two-cycles, except for divides and square-roots.

Instruction	Exceptions	Notes
LDSB	DAE	
LDSH	DAE, UA	
LDUB	DAE	
LDUH	DAE, UA	
LD	DAE, UA	
LDD	DAE, UA	
LDF	DAE, UA	
LDDF	DAE, UA	
LDFSR	FPD	
LDC	CPD	
LDDC	CPD	
LDCSR	CPD	
LDSBA	DAE, PI, II	
LDSHA	DAE, PI, II, UA	
LDUBA	DAE, PI, II	
LDUHA	DAE, PI, II, UA	
LDA	DAE, PI, II, UA	
LDDA	DAE, PI, II, UA	
STB	DAE	
STH	DAE, UA	
ST	DAE	
STD	DAE, UA	
STF	DAE, UA	
STDF	DAE, UA	
STFSR	FPD	
STDFQ	FPD	Not implemented (no FP queue)
STC	CPD	Not implemented (no CP)
STDC	CPD	Not implemented (no CP)
STCSR	CPD	Not implemented (no CP)
STDCQ	CPD	Not implemented (no CP)
STBA	DAE, PI, II	
STHA	DAE, PI, II, UA	
STA	DAE, PI, II, UA	
STDA	DAE, PI, II, UA	
LDSTUB	DAE	
LDSTUBA	DAE, PI, II	
SWAP	DAE	
SWAPA	DAE, PI, II	

Figure 2.3: Actually implemented data transfer instructions and generated exceptions

```
SETHI
NOP
SAVE        WOF
RESTORE     WUF
RDY         PI
RDASR       PI
RDPSR       PI
RDWIM       PI
RDTBR       PI
WRY         PI
WRASR       PI
WRPSR       PI, II
WRWIM       PI
WRTBR       PI
STBAR                   NOP.
UNIMP       II          Always generates Illegal instruction trap.
FLUSH                   Flushes entire ICACHE and DCACHE.
```

Figure 2.4: Actually implemented miscellaneous instructions and exceptions

```
Operation  Cycles Pipelined?
fsqrts     16      No
fsqrtd     24      No
fdivs      16      No
fdivd      24      No
Others      2*     Yes

* Add 2 cycles if denormalization of
  result is needed.
```

- Load-store operations take two cycles on hits, and 40 cycles on misses (with single cycle SRAM main memory).

```
Operation Cycles  Pipelined?
loads      2*        Yes
stores     2*        Yes
swap       8*        Yes
ldstub     8*        Yes

* If there is a hit in Cache.
  Miss-penalty to fetch cache line
  of 64 bytes is 60 cycles.
```

- Save/Restore instructions take 3 cycles to execute.

```
AND
ANDcc
ANDN
ANDNcc
OR
ORcc
ORN
ORNcc
XOR
XORcc
XNOR
XNORcc
SLL
SRL
SRA
ADD
ADDcc
ADDX
ADDXcc
TADDcc
TADDccTV    TOF
SUB
SUBcc
SUBX
SUBXcc
TSUBcc
TSUBccTV    TOF
MULScc
UMUL
SMUL
UMULcc
SMULcc
UDIV        IDZ
SDIV        IDZ
UDIVcc      IDZ
SDIVcc      IDZ
```

Figure 2.5: Integer ALU instructions and associated traps

```
TA      TT
TN      TT
TNE     TT
TE      TT
TG      TT
TLE     TT
TGE     TT
TL      TT
TGU     TT
TLEU    TT
TCC     TT
TCS     TT
TPOS    TT
TNEG    TT
TVC     TT
TVS     TT
```

Figure 2.6: TICC instructions and associated traps

2.3 AJIT thread Identification Codes

Each AJIT thread can be identified by accessing ASR 29. ASR 29 contains the following information:

```
[31:16]  0x5052
[15:8]   Core-id
[7:0]    Thread-id
```

Further, implementation details about the AJIT thread can be identified by accessing ASR 28. ASR 28 contains the following information:

```
[31:30]    L1 Dcache size
   00 -> 4KB
   01 -> 8KB
   10 -> 16KB
   11 -> 32KB
[29:28]   L1 Icache size
   00 -> 4KB
   01 -> 8KB
   10 -> 16KB
   11 -> 32KB
[27:26]   Log of dcache associativity
[25:24]   Log of icache associativity
[23:20]   Dcache hit latency
[19:16]   Icache hit latency
[15:12]   Log of MMU L3 TLB size
```

```
BA
BN
BNE
BE
BG
BLE
BGE
BL
BGU
BLEU
BCC
BCS
BPOS
BNEG
BVC
BVS

CALL
JMPL     UA
RETT     II, PI, UA

   Floating point branch.
FBA      FPD
FBN      FPD
FBU      FPD
FBG      FPD
FBUG     FPD
FBL      FPD
FBUL     FPD
FBLG     FPD
FBNE     FPD
FBE      FPD
FBUE     FPD
FBGE     FPD
FBUGE    FPD
FBLE     FPD
FBULE    FPD
FBO      FPD

   Co-processor branch
CBA      CPD      Not implemented
CBN      CPD      Not implemented
CB3      CPD      Not implemented
CB2      CPD      Not implemented
CB23     CPD      Not implemented
CB1      CPD      Not implemented
CB13     CPD      Not implemented
CB12     CPD      Not implemented
CB123    CPD      Not implemented
CB0      CPD      Not implemented
CB03     CPD      Not implemented
CB02     CPD      Not implemented
CB023    CPD      Not implemented
CB01     CPD      Not implemented
CB013    CPD      Not implemented
CB012    CPD      Not implemented
```

FiTOs	FPD	
FiTOd	FPD	
FiTOq	FPD	
FsTOi	FPD, FPE	
FdTOi	FPD, FPE	
FqTOi	FPD, FPE	Not implemented
FsTOd	FPD, FPE	
FsTOq	FPD, FPE	Not implemented
FdTOs	FPD, FPE	
FdTOq	FPD, FPE	Not implemented
FqTOs	FPD, FPE	Not implemented
FqTOd	FPD, FPE	Not implemented
FMOVs	FPD	
FNEGs	FPD	
FABSs	FPD	
FSQRTs	FPD, FPE	
FSQRTd	FPD, FPE	
FSQRTq	FPD, FPE	Not implemented
FADDs	FPD, FPE	
FADDd	FPD, FPE	
FADDq	FPD, FPE	Not implemented
FSUBs	FPD, FPE	
FSUBd	FPD, FPE	
FSUBq	FPD, FPE	Not implemented
FMULs	FPD, FPE	
FMULd	FPD, FPE	
FMULq	FPD, FPE	Not implemented
FsMULd	FPD, FPE	
FdMULq	FPD, FPE	Not implemented
FDIVs	FPD, FPE	
FDIVd	FPD, FPE	
FDIVq	FPD, FPE	Not implemented
FCMPs	FPD	
FCMPd	FPD	
FCMPq	FPD, FPE	Not implemented
FCMPEs	FPD, FPE	
FCMPEd	FPD, FPE	
FCMPEq	FPD, FPE	Not implemented

Figure 2.8: Floating point instructions and associated traps

CPop1	CPD	Not implemented
CPop2	CPD	Not implemented

Figure 2.9: Coprocessor instructions and associated traps

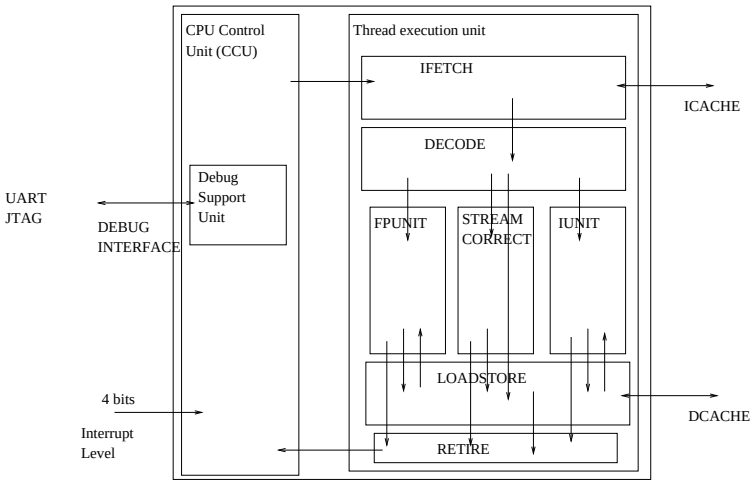

Figure 2.10: 32-bit AJIT CPU (Thread) Internal Structure

```
[11:9]    Log of MMU L2 TLB size
[8:7]     Log of MMU L1 TLB size
[6:5]     Log of MMU L0 TLB size
[4:3]     Unused.
[2]       Has NC bypass path.
[1]       2-thread/1-thread.
[0]       set if 64-bit isa else 32-bit isa
```

2.3.1 AJIT CPU (Thread) resets and mode signal

The AJIT thread has an 8-bit reset input which puts the thread into various operating modes. The 8-bit reset [7:0] is interpreted as follows

```
[7]  unused
[6]  unused
[5]  unused
[4]  in debug mode, communicate each trap to
          remote debugger
[3]  enable trace logging
[2]  start thread in single-step mode
[1]  start thread in debug mode
[0]  thread reset (pc, npc, psr initialized)
```

The processor indicates which mode it is in using a 2-bit mode signal. The mode signal [1:0] is to be interpreted as follows:

```
00  post power on reset
01  post thread reset (pc, npc, psr initialized)
```

```
10  run mode (after thread reset is released)
11  error mode (halted)
```

Typically, the reset sequence is as follows

```
power-on-reset -> thread-reset -> thread-reset-release
```

Assertion of the power-on-reset puts the processor into mode 0. In state 0, assertion of the thread reset (bit 0) puts the processor into mode 1. In mode 1, release of the thread reset puts the processor into mode 2 (the run mode). If the processor goes encounters an error condition, it goes into mode 3 from which it can be removed only by a power-on-reset.

Chapter 3

64-bit ISA extensions and the AJIT CPU64 execution thread

The AJIT CPU32 processor implements the Sparc-V8 ISA. We have extended this ISA to provide support for a native 64-bit integer datatype. The extensions use the existing instruction encodings, and the additional instructions use the unused encodings in the SPARC-V8 ISA. The resulting CPU with ISA extensions is called the AJIT CPU64.

All extensions are RegisterXRegister -¿ Register,Condition-codes type instructions. The load/store instructions are not modified.

We list the additional instructions in the subsequent sections. In each case, only the differences in the encoding relative to an existing Sparc-V8 instruction are provided.

3.1 Integer-unit extensions: Arithmetic-logic instructions

These instructions provide 64-bit arithmetic/logic support in the integer unit. The instructions work on 64-bit register pairs in most cases. Register-pairs are identified by a 5-bit even number (lowest bit must be 0).

The instructions function the same as ADD/SUB/UMUL/UDIV/SMUL/SDIV and their CC versions. The trapping behaviour of the *TV instructions is not extended to 64-bit values.

The *MULSCC operation is not extended to 64 bits, because we expect to have a divider in all implementations.

```
ADDD
  encoding:  same as ADD, but with Instr[13]=0 (i=0), and Instr[5]=1.
```

```
  rd(pair) <- rs1(pair) + rs2(pair)

ADDDCC
  encoding:  same as ADDCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) + rs2(pair), set C,O,Z,N

SUBD
  encoding:  same as SUB, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) - rs2(pair)

SUBDCC
  encoding:  same as SUBCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) - rs2(pair), set C,O,Z,N

SLLD
  encoding:  same as SLL, but with Instr[7:6]=2.
    if imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount.
    else shift-amount is the lowest 6 bits of rs2. Note that rs2
    is a 32-bit register.
  rd(pair) <-  rs1(pair) << shift-amount

SRLD
  encoding:  same as SRL, but with Instr[7:6]=2.
    if imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount.
    else shift-amount is the lowest 6 bits of rs2. Note that rs2
    is a 32-bit register.
  rd(pair) <-  rs1(pair) >> shift-amount

SRAD
  encoding:  same as SRA, but with Instr[7:6]=2.
    if imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount.
    else shift-amount is the lowest 6 bits of rs2. Note that rs2
    is a 32-bit register.
  rd(pair) <-  rs1(pair) >> shift-amount (with sign extension).

UMULD
  encoding:  same as UMUL, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) * rs2(pair)
UMULDCC
  encoding:  same as UMULCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) * rs2(pair), sets Z, Ovflow

SMULD
  encoding:  same as SMULD, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) * rs2(pair) (signed)
SMULDCC
```

```
  encoding:  same as SMULCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) * rs2(pair) (signed)
  sets condition codes Z,N,Ovflow

UDIVD
  encoding:  same as UDIV, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) / rs2(pair)
    note: can generate div-by-zero trap.
UDIVDCC
  encoding:  same as UDIVCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) / rs2(pair)
    sets condition codes Z,Ovflow
    note: can generate div-by-zero trap.
SDIVD
  encoding:  same as SDIV, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) / rs2(pair) (signed)
SDIVDCC
  encoding:  same as SDIVCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) / rs2(pair) (signed)
    sets condition codes Z,N,Ovflow
    note: can generate div-by-zero trap.


ORD
  encoding:  same as OR, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) | rs2(pair)
ORDCC
  encoding:  same as ORCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) | rs2(pair), sets Z.
ORDN
  encoding:  same as ORN, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) | (~rs2(pair))
ORDNCC
  encoding:  same as ORNCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) | (~rs2(pair)), sets Z
  sets Z.

XORD
  encoding:  same as XOR, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) ^ rs2(pair)
XORDCC
  encoding:  same as XORCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) ^ rs2(pair), sets Z
  sets Z.

XNORD
```

```
  encoding:  same as XNOR, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) ^ rs2(pair)
XNORDCC
  encoding:  same as XNORCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) ^ rs2(pair), sets Z

ANDD
  encoding:  same as AND, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) . rs2(pair)
ANDDCC
  encoding:  same as ANDCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) . rs2(pair), sets Z
ANDDN
  encoding:  same as ANDN, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd(pair) <- rs1(pair) . (~rs2(pair))
ANDDNCC
  encoding:  same as ANDNCC, but with Instr[13]=0 (i=0), and Instr[5]=1.
  rd <- rs1 . (~rs2), sets Z
```

3.2 Integer-unit extensions: SIMD (Vector) instructions

These instructions are vector instructions which work on two source registers (each a 64 bit register pair), and produce a 64-bit vector result. The vector elements can be 8-bit/16-bit/32-bit.

```
VADDD8, VADDD16, VADDD32
  encoding:  same as ADDD, but with Instr[13]=0 (i=0), and Instr[6:5]=2.
       bits Instr[9:7] are a 3-bit field, which specify the data
       type
      001   byte      (VADDD8)
      010   half-word (16-bits)  (VADDD16)
      100   word (32-bits)     (VADDD32)
  performs a vector operation by considering the 64-bit operands as
  a vector of objects with specified data-type.

  vadd8 rs1,rs2, rd
  vadd16
  vadd32

VSUBD8, VSUBD16, VSUBD32
  encoding:  same as SUBD, but with Instr[13]=0 (i=0), and Instr[6:5]=2.
       bits Instr[9:7] are a 3-bit field, which specify the data
       type
      001   byte       (VSUBD8)
```

```
        010   half-word (16-bits)  (VSUBD16)
        100   word (32-bits)     (VSUBD32)
    performs a vector operation by considering the 64-bit operands as
    a vector of objects with specified data-type.

    vsubd8
    vsubd16
    vsubd32

  VUMULD8, VUMULD16, VUMULD32
    encoding:  same as UMULD, but with Instr[13]=0 (i=0), and Instr[6:5]=2.
         bits Instr[9:7] are a 3-bit field, which specify the data
         type
        001   byte      (VMULD8)
        010   half-word (16-bits)  (VMULD16)
        100   word (32-bits)     (VMULD32)
    performs a vector operation by considering the 64-bit operands as
    a vector of objects with specified data-type.

    vumuld8
    vumuld16
    vumuld32

  VSMULD8, VSMULD16, VSMULD32
    encoding:  same as SMULD, but with Instr[13]=0 (i=0), and Instr[6:5]=2.
         bits Instr[9:7] are a 3-bit field, which specify the data
         type
        001   byte      (VSMULD8)
        010   half-word (16-bits)  (VSMULD16)
        100   word (32-bits)     (VSMULD32)
    performs a vector operation by considering the 64-bit operands as
    a vector of objects with specified data-type.

    vsmuld8
    vsmuld16
    vsmuld32
```

3.3 Integer-unit extensions: SIMD REDUCE instructions

These instructions are vector instructions which reduce a 64 bit source register to a destination using an associative operation.

```
  ADDDREDUCE8
    op=2, op3[3:0]=0xd, op3[5:4]=0x2, contents[7:0] of rs2 specify a mask.
```

```
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    32-bit register.
  Instr[24:19] (op3) = 101101
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  = 0 (ignored)
  Instr[12:10]   (zero)
  Instr[9:7]
    = 1 for byte reduce
      contents[7:0] of rs2 specify a mask.
  Instr[6:5]  (zero)
  Instr[4:0]   (rs2)   32-bit register is read.

  rd <- (m7 ? rs1_7 : 0x0) + (m6 ? rs1_6 : 0x0) + (m5 ? rs1_5:0)  ...
        + (m0 ? rs1_0 : 0x0)


  adddreduce8 %rs1, %rs2,  %rd
    (all additions are twos-complement, 8-bit additions)

ADDDREDUCE16
  op=2, op3[3:0]=0xd, op3[5:4]=0x2, contents[3:0] of rs2 specify a mask.
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    32-bit register.
  Instr[24:19] (op3) = 101101
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  = 0 (ignored)
  Instr[12:10]   (zero)
  Instr[9:7]   = 2 for half word reduce
      contents[3:0] of rs2 specify a mask.
  Instr[6:5]  (zero)
  Instr[4:0]   (rs2)   32-bit register is read.

  rd <- (m3 ? rs1_hw_3 : 0x0) + (m2 ? rs1_hw_2 : 0x0)
      + (m1 ? rs1_hw_1: 0x0) + (m0 ? rs1_hw_0 : 0x0)


  adddreduce16 %rs1, %rs2,  %rd
    (all additions are twos-complement, 16-bit additions)

ORDREDUCE8
  op=2, op3[3:0]=0xe, op3[5:4]=0x2, contents[7:0] of rs2 specify a mask.
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    rd is a 32-bit register.
```

```
  Instr[24:19] (op3) = 101110
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  = 0 (ignored)
  Instr[12:10]   (zero)
  Instr[9:7]
    = 1 for byte reduce
      contents[7:0] of rs2 specify a mask.
  Instr[6:5]  (zero)
  Instr[4:0]   (rs2)   32-bit register is read.

  rd <- (m7 ? rs1_7 : 0x0) | (m6 ? rs1_6 : 0x0) | (m5 ? rs1_5:0)  ...
        | (m0 ? rs1_0 : 0x0)

  ordreduce8 %rs1, %rs2,  %rd

ORDREDUCE16
  op=2, op3[3:0]=0xe, op3[5:4]=0x2, contents[3:0] of rs2 specify a mask.
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    rd is a 32-bit register.
  Instr[24:19] (op3) = 101110
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  = 0 (ignored)
  Instr[12:10]   (zero)
  Instr[9:7]
    = 2 for half-word reduce
      contents[3:0] of rs2 specify a mask.
  Instr[6:5]  (zero)
  Instr[4:0]   (rs2)   32-bit register is read.

  rd <- (m3 ? rs1_hw_3 : 0x0) | (m2 ? rs1_hw_2 : 0x0)
      | (m1 ? rs1_hw_1: 0x0) | (m0 ? rs1_hw_0 : 0x0)

  ordreduce16 %rs1, %rs2,  %rd


ANDDREDUCE8
  op=2, op3[3:0]=0xf, op3[5:4]=0x2, contents[7:0] of rs2 specify a mask.
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    rd is a 32-bit register.
  Instr[24:19] (op3) = 101111
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  = 0 (ignored)
  Instr[12:10]   (zero)
  Instr[9:7]
```

```
    = 1 for byte reduce
      contents[7:0] of rs2 specify a mask.
  Instr[6:5]  (zero)
  Instr[4:0]   (rs2)   32-bit register is read.
  rd <- ( (m7 ? rs1_7 : 0xff) . (m6 ? rs1_6 : 0xff) .... (m0 ? rs1_0 : 0xff))

  anddreduce8 %rs1, %rs2,  %rd

ANDDREDUCE16
  op=2, op3[3:0]=0xf, op3[5:4]=0x2, contents[3:0] of rs2 specify a mask.
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    rd is a 32-bit register.
  Instr[24:19] (op3) = 101111
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  = 0 (ignored)
  Instr[12:10]   (zero)
  Instr[9:7]
    = 2 for half-word reduce
      contents[3:0] of rs2 specify a mask.
  Instr[6:5]  (zero)
  Instr[4:0]   (rs2)   32-bit register is read.

  rd <- (m3 ? rs1_hw_3 : 0x0) + (m2 ? rs1_hw_2 : 0x0)
      + (m1 ? rs1_hw_1: 0x0) + (m0 ? rs1_hw_0 : 0x0)

  anddreduce16 %rs1, %rs2, %rd

XORDREDUCE8
  op=2, op3[3:0]=0xe, op3[5:4]=0x3, contents[7:0] of rs2 specify a mask.
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    rd is a 32-bit register.
  Instr[24:19] (op3) = 111110
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  = 0 (ignored)
  Instr[12:10]   (zero)
  Instr[9:7]
    = 1 for byte reduce
      contents[7:0] of rs2 specify a mask.
  Instr[4:0]   (rs2)   32-bit register is read.
  Instr[6:5]     (zero)
  rd <- (m7 ? rs1_7 : 0) ^ (m6 ? rs1_6 : 0) ^ (m5 ? rs1_5:0)  ...
        ^ (m0 ? rs1_0 : 0)
  xordreduce8 %rs1, %rs2,  %rd
```

```
XORDREDUCE16
  op=2, op3[3:0]=0xe, op3[5:4]=0x3, contents[3:0] of rs2 specify a mask.
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    rd is a 32-bit register.
  Instr[24:19] (op3) = 111110
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  = 0 (ignored)
  Instr[12:10]   (zero)
  Instr[9:7]
    = 2 for half-word reduce
      contents[3:0] of rs2 specify a mask.
  Instr[4:0]   (rs2)   32-bit register is read.
  Instr[6:5]     (zero)

  rd <- (m3 ? rs1_hw_3 : 0) ^ (m2 ? rs1_hw_2 : 0)
        ^ (m1 ? rs1_hw_1:0) ^ (m0 ? rs1_hw_0 : 0)

  xordreduce16 %rs1, %rs2,  %rd

ZBYTEDPOS
  op=2, op3[3:0]=0xf, op3[5:4]=0x3, contents[7:0] of rs2/imm-value specify a mask.
  encoding
  Instr[31:30] (op) = 0x2
  Instr[29:25] (rd)    rd is a 32-bit register.
  Instr[24:19] (op3) = 111111
  Instr[18:14] (rs1)   lowest bit assumed 0.
  Instr[13]    (i)  =  if 0, use rs2, else Instr[7:0]
  Instr[12:5]  = 0  (ignored if i=0)
  Instr[4:0]   (rs2, if i=0)
      32-bit register is read.

  rd <- [b7_zero b6_zero b5_zero b4_zero .. b0_zero]
    (if mask-bit is zero then b*_zero is zero)

  zbytedpos %rs1, %rs2/imm, %rd
```

3.4 Vector floating point instructions

These are vector float operations which work on two single precision operand pairs to produce two single precision results.

```
// SIMD float ops.
//   NaN propagated, but no traps.
//   For each of these, rs1,rs2,rd are
```

```
//   considered even numbers pointing to
//   a floating point register-pair.
//
VFADD32
  op=2, op3=0x34, opf=0x142
  vfadd32 %f0, %f2, %f4

VFADD16
  // half-precision vector add.
  // opcode to be assigned.
  op=2, op3=0x34, opf=0x143
  vfadd16 %f0, %f2, %f4

VFSUB32
  op=2, op3=0x34, opf=0x144
  vfsub32 %f0, %f2, %f4

VFSUB16
  op=2, op3=0x34, opf=0x145
  // half-precision vector sub.
  // opcode to be assigned.
  vfsub16 %f0, %f2, %f4

VFMUL32
  op=2, op3=0x34, opf=0x146
  vfmul32 %f0, %f2, %f4

VFMUL16
  op=2, op3=0x34, opf=0x147
  // half-precision vector multiply..
  // opcode to be assigned.
  vfmul16 %f0, %f2, %f4

  // these are vector convert instructions
  // to go from i16 to half and vice-versa.
VFI16TOH
  op=2, op3=0x34, opf=0x148
  // convert 4x i16 to 4x half-precision.
  //     rs2-pair is the input, rd-pair is the output
  vfi16toh %f0, %f2

VFHTOI16
  op=2, op3=0x34, opf=0x149
  // convert 4x half-precision to 4x i16.
  //     rs2-pair is the input, rd-pair is the output
  vfhtoi16 %f0, %f2
```

3.5 Floating point REDUCE

These act on a vector and produce a scaler result (as in the integer case).

```
FADDREDUCE16
  op=2, op3=0x34, opf=0x150
  //
  // add the four half-precision numbers in
  // the 64-bit FP register pair rs2, and produce a
  // half-precision result into the destination 32-bit FP register rd.
  //
  // Note: rs1 value is ignored.
  //
```

3.6 Half precision conversion instructions

These instructions allow conversion between IEEE half-precision numbers and IEEE single/double precision numbers and integers.

```
FSTOH
  op=2, op3=0x34, opf=0x151
  // convert single precision value to half-precision value.
  // %f1 -> %f2  (32-bit FP registers)
  //     rs2 is the input, rd is the output
  fstoh %f1, %f2
FHTOS
  op=2, op3=0x34, opf=0x152
  // convert half precision value to single-precision value.
  //     rs2 is the input, rd is the output
  // %f1 -> %f2  (32-bit FP registers)
  fhtos %f1, %f3
```

Note that the double-to-half and half-to-double, int-to-half and half-to-int instructions are not provided. This is because, these transformations are likely to be rarer. Also, the FDTOS, FDTOI, FITOS, FITOD instructions together with the added FSTOH, FHTOS instructions are sufficient (at a minor cost).

3.7 CSWAP insruction

The Sparc-V8 ISA does not include a compare-and-swap (CAS) instruction which is very useful in achieving consensus among distributed agents when the number of agents is ¿ 2.

We introduce a CSWAP instruction in two flavours

```
CSWAP     rs1, rs2/immediate, rd
  op=3
```

```
    op3= 10 1111
      (rest of instruction similar to SWAP)

    cswap %rs1, %rs2/imm, %rd

  CSWAPA    rs1, rs2/immediate, rd-pair, asi
    op=3
    op3= 11 1111
      (rest of instruction similar to SWAPA)

    cswapa %rs1, %rs2/imm, %rd with asi specified.
```

The semantics of the instruction (the entire sequence is atomic) are as follows:

```
TMPVAL = mem[rs1]  (load word, lock system bus)
if <rs2/immediate> == TMPVAL
  (store, unlock) mem[rs1] = <rd>
  <rd>  = TMPVAL
else
  (store, unlock) mem[rs1] = TMPVAL
```

The store under else is redundant but is required in order to unlock the bus. In the same way as in the SWAP instruction

- mem[rs1] is left either with its value prior to the instruction or with the value in rd.
- ¡rd¿ is left either with its value prior to the instruction or with the value in mem[rs1].

The processor can check rd after execution to confirm if the swap succeeded.

3.8 Timings of the additional instructions in the AJIT CPU64 execution thread

The additional instructions in the integer unit are all single cycle instructions, with the exception of the divide instruction. The additional floating point instructions are all pipelined two cycle instructions. The compare and swap instruction takes multiple cycles because it involves a memory load and a memory store.

Chapter 4

The AJIT memory unit

The AJIT memory unit consists of two VIVT caches for data and instruction access, and a shared memory management unit, as shown in Figure 4.1.

4.1 The AJIT Data Cache

The AJIT data cache (DCACHE) is a pipelined implementation which supports precise exceptions. The DCACHE has a cache line size of 64 bytes, and is nominally configured to a size of 32KB, although it can be scaled to sizes down to 4KB.

Two implementations of the DCACHE are available:

- A direct-mapped write-through-allocate virtually-indexed-virtually-tagged (VIVT) cache. The direct mapped version has a hit latency of two clock cycles.
- A 2-way/4-way/8-way set-associative write-through-allocate virtually-indexed-virtually-tagged (VIVT) cache. The set associative version has a hit latency of two or three clock cycles. On average, the associative cache has a hit latency which is 3% higher than that of the direct mapped cache.
- The miss latency of the caches (assuming SRAM main memory) is approximately 36 clock cycles.

Further, the DCACHE has a couple of optimizations.

- The DCACHE provides a fully pipelined bypass access path. Thus, load/store bypass accesses which use ASI values between 0x20 and 0x2f are directly passed through to the system bus, and executed in pipelined fashion. This makes communication between the CPU and peripheral device memory substantially faster.
- The DCACHE provides a fully pipelined access path for accesses to non-cacheable memory locations. The DCACHE maintains a 4-entry TLB

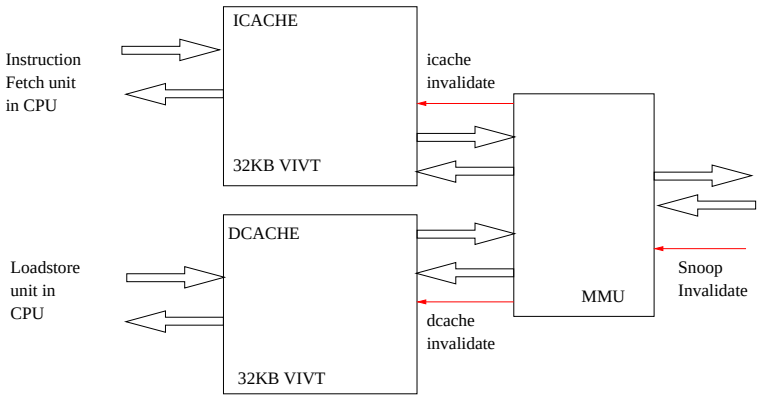

Figure 4.1: AJIT Memory Unit

which keeps track of recently accessed pages which are known to be non-cacheable.

4.2 The AJIT Instruction Cache

The AJIT instruction cache (ICACHE) is a pipelined implementation which supports precise exceptions. The ICACHE has a cache line size of 64 bytes, and is nominally configured to a size of 32KB, although it can be scaled to sizes down to 4KB.

Two implementations of the ICACHE are available:

- A direct-mapped write-through-allocate virtually-indexed-virtually-tagged (VIVT) cache. The direct mapped version has a hit latency of two clock cycles.
- A 2-way/4-way/8-wayset-associative write-through-allocate virtually-indexed-virtually-tagged (VIVT) cache. The set associative version has a hit latency of either two or three clock cycles. On average, the set associative version has a hit latency which is 3% higher than the direct mapped version.

4.3 The AJIT Memory Management Unit (MMU)

The SPARC reference MMU is implemented. The MMU contains four distinct TLB's as shown below:

```
2-entry   Level-0 TLB
4-entry   Level-1 TLB
16-entry  Level-2 TLB
256-entry Level-3 TLB
```

The following ASI's are required to be supported, and are in fact supported.

```
ASI    Description
------------------------------
0x3    MMU-flush-probe
         flush => whole MMU TLB is flushed
         probe => only entire probes are supported.

0x4     MMU-register-access
0x8     User-instruction-fetch
0x9     Supervisor-instruction-fetch
0xa     User-data-access
0xb     Supervisor-data-access
0x20-0x2f
        MMU bypass
0x30    MMU bypass (reserved for future use)
```

Bit 8 of the MMU control register is used to encode a default cacheable bit. When the MMU is disabled, bit-8 is passed to the caches as the cacheable bit for the fetched line. This allows the caches to be used even when the MMU is disabled.

Note that the use of this cacheable bit is not standard. The standard behaviour as specified by the SPARC V8 SRMMU specification is that when the MMU is disabled, all accesses are marked non-cacheable. The non-standard extension allows us to disable the MMU functionality while maintaining the use of the caches. Such a capability is useful in evaluating a system which excludes the MMU entirely.

4.4 The AJIT Level-2 Cache

The AJIT level-2 is a physically indexed, physically tagged, 8-way set associative, write-back cache that can be configured for sizes of 64KB to 1MB. The L2 cache has a line size of 64 bytes.

The L2 cache can be used in a flexible manner for assembling AJIT processor systems.

4.5 Cache coherence

In a single core AJIT processor, there is no explicit coherence maintained between the data and instruction caches. Thus, if you plan to use self modifying code, you will need to issue a flush instruction in order to get the modified code into the instruction cache.

In a multi-core AJIT processor, the coherence between multiple caches across cores is maintained by using a snoop protocol with invalidation on writes. The multi-core memory controller for the AJIT processor ensures that total store

ordering is maintained, and generates invalidate messages in the order that writes are applied at the memory. The controller also uses a snoop filter to reduce traffic between the cores and the memory controller.

Chapter 5

The AJIT Multi-threaded Core

The AJIT Multi-threaded core is constructed using one or two AJIT CPU's, which work together with a shared memory unit.

5.1 The AJIT single-thread core

The single-thread core is illustrated in Figure 5.1. As indicated, the core integrates an AJIT CPU (either AJIT32 CPU or AJIT64 CPU) with the memory unit. The AJIT CPU has its own debug interface and its own interrupt input. This is the minimal AJIT core configuration.

5.2 The AJIT dual-thread core

The dual-thread core is illustrated in Figure 5.2. As indicated, the core integrates two identical AJIT CPU's (either AJIT32 CPU or AJIT64 CPU) with a shared memory unit. Each AJIT CPU has its own debug interface and its own interrupt input and work indendependently.

In the dual threaded core, each thread includes a 1KB instruction buffer in the instruction access path between the thread and the instruction cache. This helps reduce the contention for the instruction cache, especially when one of the threads is executing tight loops.

5.3 The AJIT lite core

The single-thread AJIT lite core omits the floating point unit and the memory management unit from the single threaded core in Figure 5.1. This results in a 50% saving of resources over the minimal single threaded core.

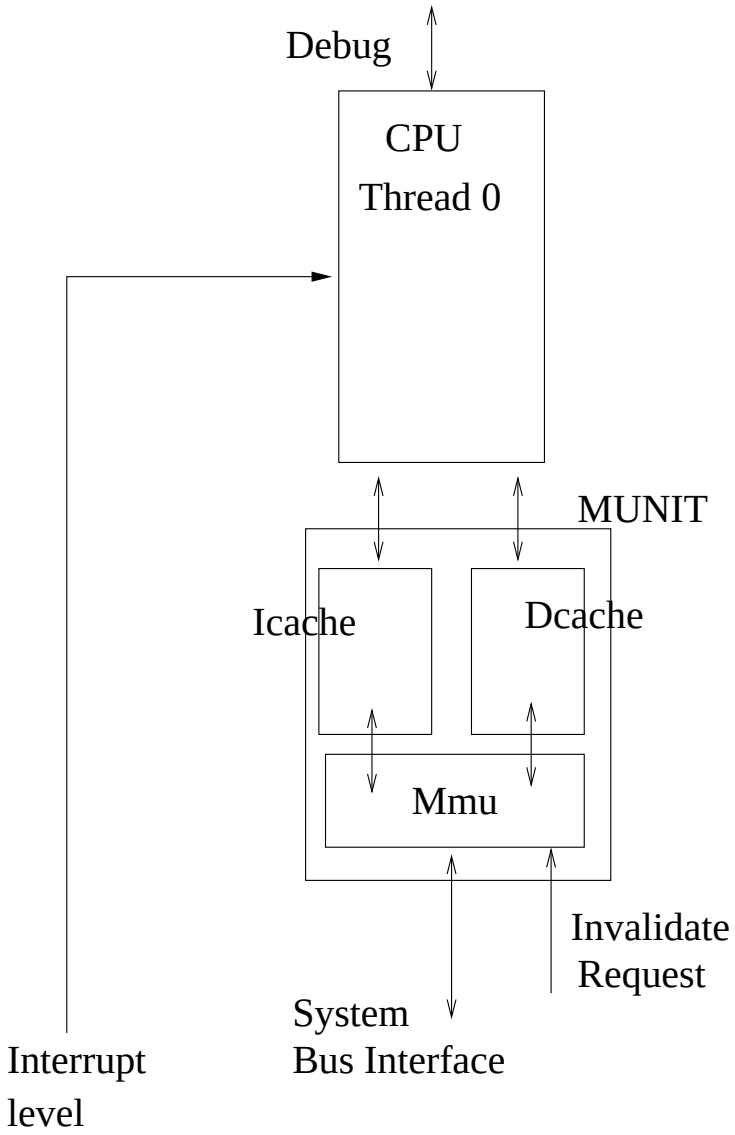

Figure 5.1: AJIT single-threaded core

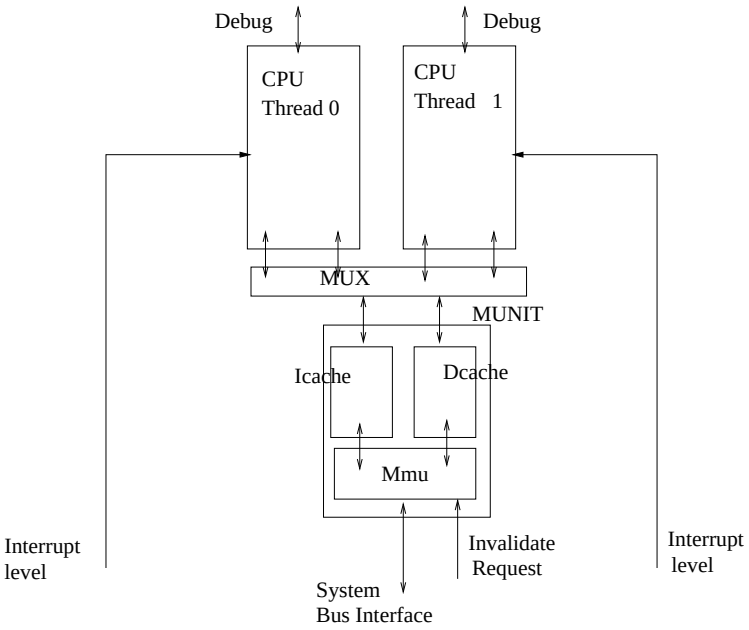

Figure 5.2: AJIT dual-threaded core

Chapter 6

AJIT Multi-core Processors

Using multiple cores, we construct AJIT coherent shared memory multi-core implementations. An example of a quad-core AJIT processor is shown in Figure 6. The four cores (which may be of any of the four types 1x32, 2x32, 1x64, 1x2x64) are connected to a coherent memory controller which uses a snoop protocol to invalidate cache entries on a write. The interrupt interfaces on the individual CPU's and the debug interfaces to the CPU's in the system are also shown in the figure. The range of available multi-cores ranges from the 2x1x32 configuration to the 4x2x64 configuration. The 4x2x64 configuration has been validated on FPGA.

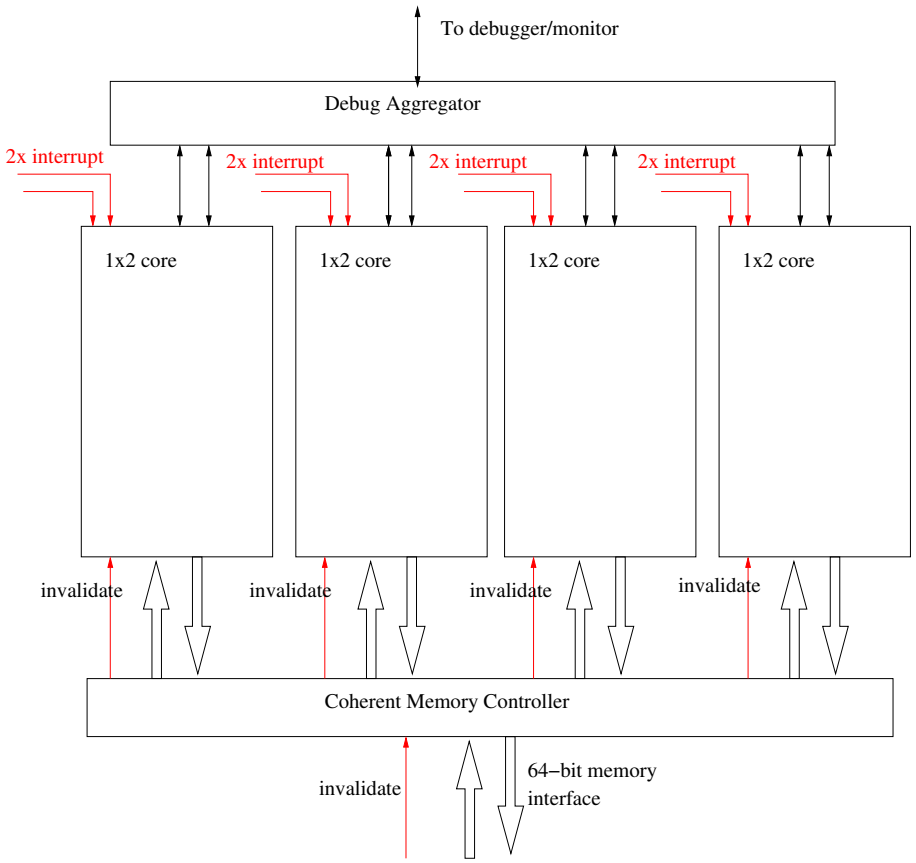

Figure 6.1: AJIT multi-core with four $1x2$ cores and coherent memory controller

Chapter 7

AJIT processor interfaces

The primary interfaces between the AJIT processor and the external world are:

- AJIT CORE BUS interface (ACB): A 64-bit data, 36-bit address system memory interface.
- AJIT FIFO BUS interface (AFB): A 32-bit data, 36-bit peripheral/memory interface.

Both these bus interfaces involve a request and a response interface. The bus master sends a request word to the slave, which responds to the request with a response word. The exchange of data between the master and slave is regulated using a simple two-wire protocol.

7.1 AJIT core bus interface

This consists of a request FIFO and a response FIFO. The requester (AJIT processor) writes a request word into the request FIFO. The 110-bit request word has the following format:

```
--------------------------------
bit-field          meaning
--------------------------------
[109]              lock
[108]              read/write_bar
[107:100]          byte mask
[99:64]            address
[63:0]             write-data
--------------------------------
```

The response FIFO consists of a 65-bit word with the following format:

```
--------------------------------
bit-field          meaning
```

[64]	error
[63:0]	read-data

The request/response pair can be used with a pipelined memory.

7.2 AJIT FIFO bus interface

This also consists of a request FIFO and a response FIFO. The requester (AJIT processor) writes a request word into the request FIFO. The 74-bit request word has the following format:

bit-field	meaning
[73]	lock
[72]	read/write_bar
[71:68]	byte mask
[67:32]	address
[31:0]	write-data

The response FIFO consists of a 33-bit word with the following format:

bit-field	meaning
[32]	error
[31:0]	read-data

The request/response pair can be used with a pipelined memory.

7.3 ACB/AFB Protocol and Timing

Both the ACB and AFB schemes are similar in their timing. We illustrate the timing based on the setup described in Figure 7.1.

1. The master initiates a transaction by sending request data to the slave. Thus, the first transfer in a transaction is a request word (110 bits for ACB, 74 bits for AFB) which is transferred from the master to the slave.

2. After the slave has computed the response to the request, the slave sends response data to the master. Thus, the second transfer is response word (65 bits for ACB, 33 bits for AFB) which is sent to the master.

The protocol and protocol for each transfer is similar and is described below.

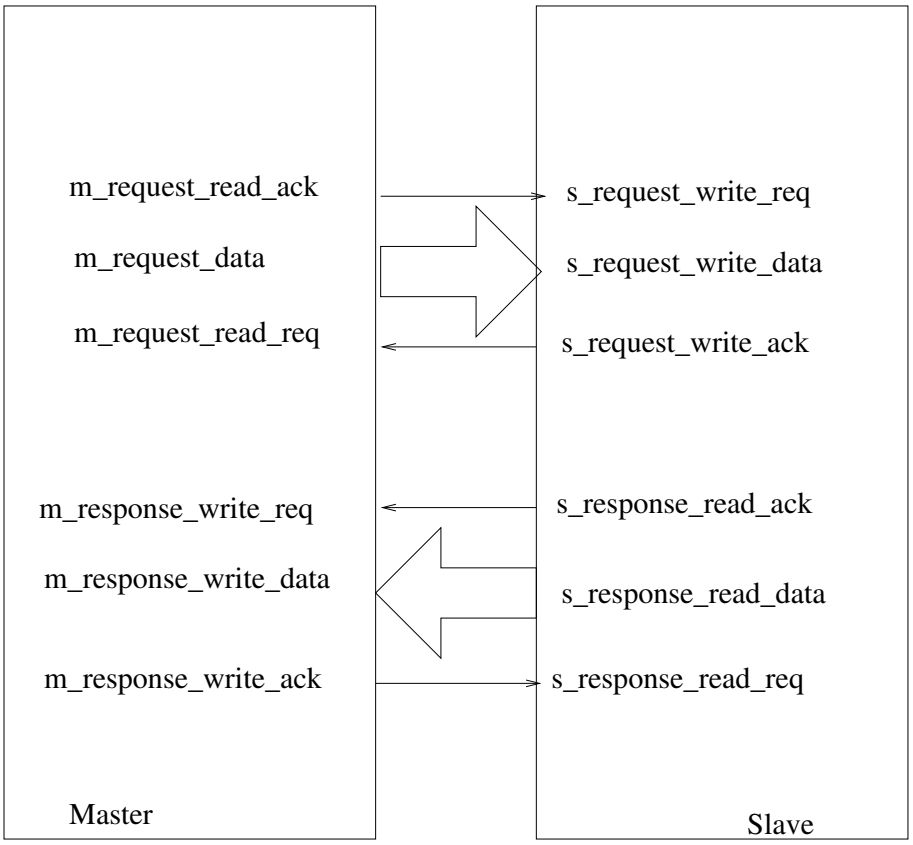

Figure 7.1: Visualization of setup for ACB/AFB master slave connections

7.3.1 Timing of a single transfer

The protocol relies to two wires between the sender and receiver.

- The first wire goes from the sender to the receiver. On the sender side, this wire is interpreted as *sender has data to send*. This wire connects to the receiver's write_req port and the senders read_ack port.
- The second wire goes from the receiver to the sender. On the receiver side, this wire is interpreted as *receiver ready to receive*. This wire connects to the sender's read_req port and to the receiver's write_ack port.

Now the behaviour of the sender can be summarized as follows:

```
sender_has_data_to_send = 0
if (sender wants to send) {
   sender_data = DATA,
      sender_has_data_to_send = 1
   while(1) {
      if (receiver_ready_to_receive)
         break; /* receiver has accepted */
   }
}
```

The behaviour of the receiver can be summarized as follows:

```
receiver_ready_to_receive = 0
if (receiver wants to receive) {
   receiver_ready_to_receive = 1
   while(1) {
      if (sender_has_data_to_send)
         REGISTER = sender_data
         break; /* receiver has accepted */
   }
}
```

This protocol is highly flexible and allows either the sender or receiver to block the transaction. For example, if the sender is faster than the receiver, the timing diagram will be as shown in Figure 7.2. If the receiver is faster than the sender, the timing diagram will be as shown in Figure 7.3. If both sender and receiver are full-rate, then the timing diagram will be as shown in Figure 7.4.

7.4 Summary

The ACB and AFB bus interfaces offer a simple, flexible mechanism to connect components in an AJIT system. Each bus interface consists of a request connection and a response connection. The data transfer on the request and response connection is managed using a simple two-wire protocol which can be used for full rate transfers as well as slow transfers.

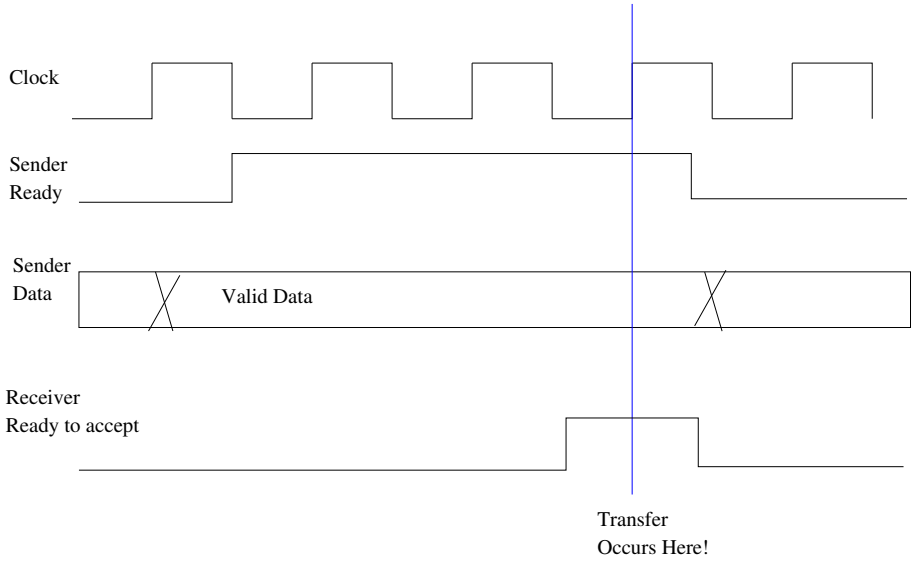

Figure 7.2: Timing Diagram for fast sender, slow receiver

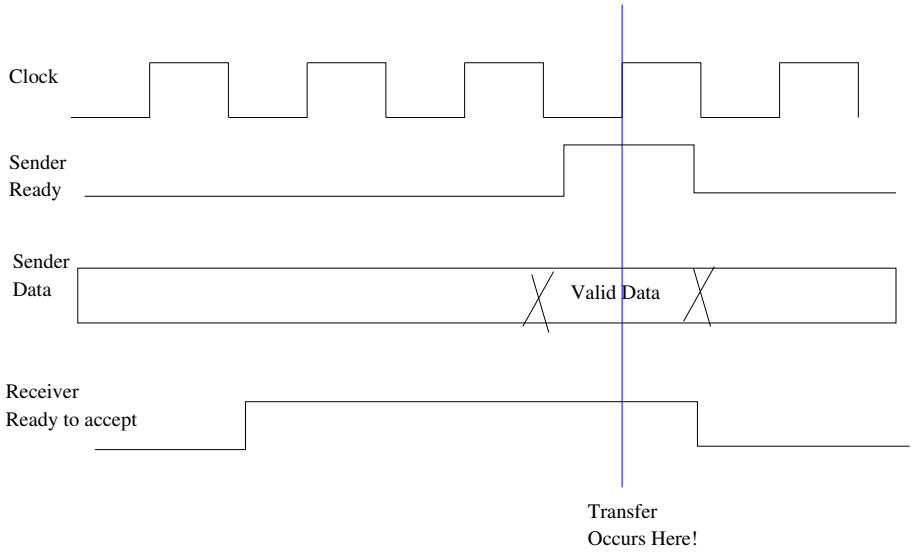

Figure 7.3: Timing Diagram for slow sender, fast receiver

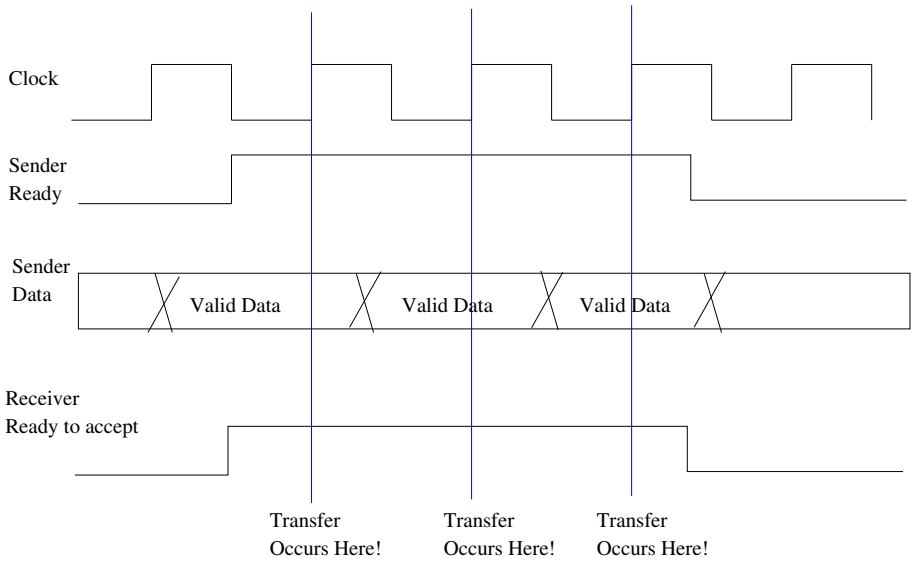

Figure 7.4: Timing Diagram for full-rate sender and receiver

Chapter 8

Basic peripherals available for AJIT systems

Basic peripherals that can be used in the FPGA are available. These include: a timer, a UART, an interrupt controller, an SPI flash controller, an I2C master, an SPI master.

8.1 The AJIT multi-core interrupt controller

The AJIT multi-core interrupt controller is implemented as shown in Figure 8.1.

The AJIT multi-core interrupt controller is a collection of (up to) eight thread interrupt controllers (TIC). Each thread interrupt controller corresponds to an active CPU thread in the AJIT multi-core system, and is identified by a (core_id, thread_id) pair, where core_id is either 0/1/2/3 and thread_id is 0/1. With the TIC complex, it is possible to dynamically map interrupts to threads, with complete flexibility.

The inter-processor-interrupt (IPI) mechanism allows a running to thread to set an interrupt to any other thread. This IPI interrupt uses interrupt level 13. The IPI is defined by two 32-bit registers: an IPI mask and an IPI value. The IPI interrupt to a thread is generated by AND-ing the IPI mask bit and the IPI value for the specific thread index.

8.1.1 Registers and their memory mapping

The addressable registers inside the interrupt controller are classified into the following groups

- The TIC registers.
- The IPI registers.

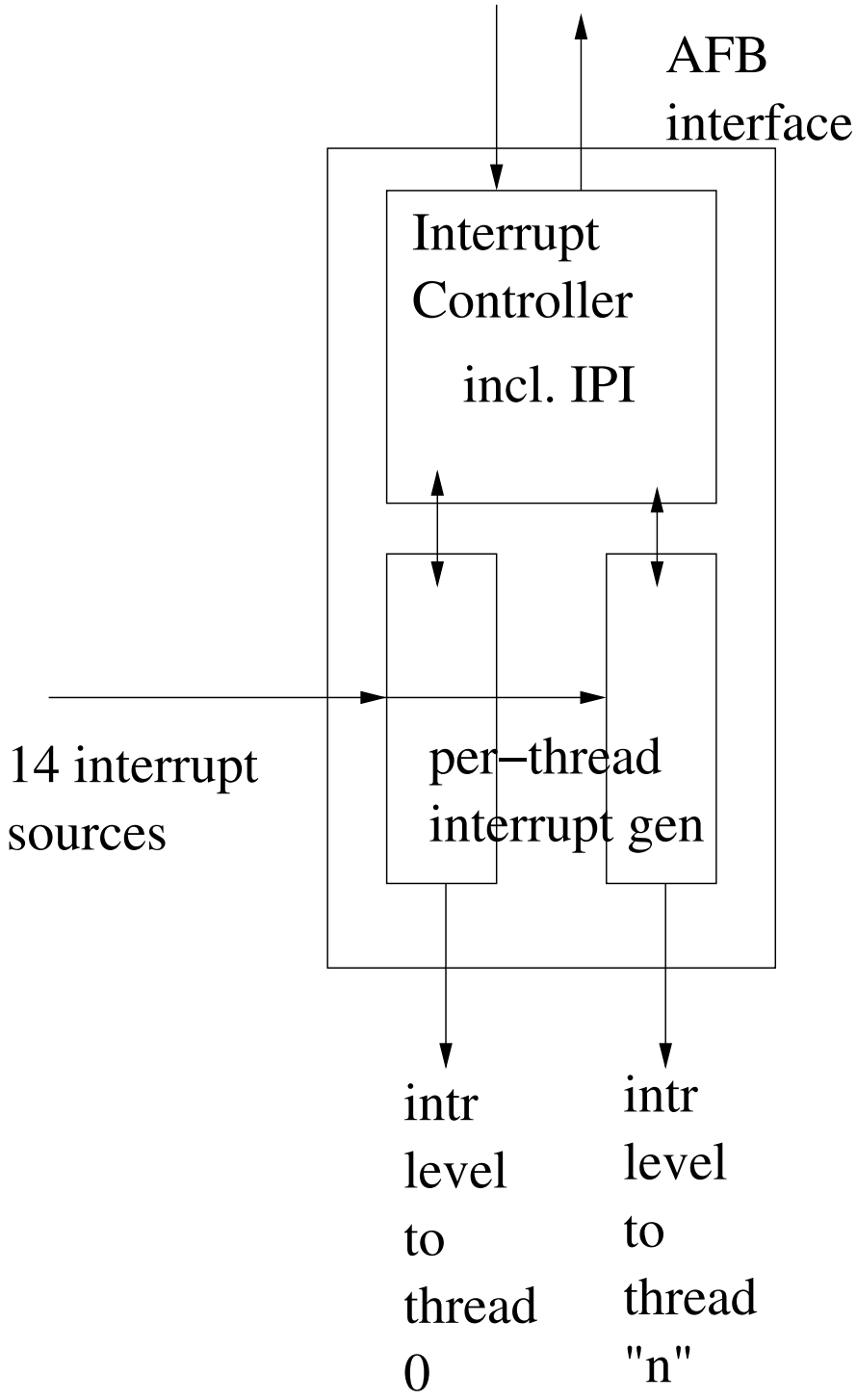

Figure 8.1: AJIT multi-core capable interrupt controller structure

Each TIC has one register assigned to it. The addresses of these registers are as follows:

```
-------------------------------------
core-id thread-id    register-address
-------------------------------------
0         0          BASE
0         1          BASE  + 0x4
1         0          BASE  + 0x8
1         1          BASE  + 0xc
2         0          BASE  + 0x10
2         1          BASE  + 0x14
2         0          BASE  + 0x18
2         1          BASE  + 0x1c
-------------------------------------
BASE = base address of the interrupt
       controller (0xFFFF3000)
```

The control register for each TIC has the following format

```
[31:17] interrupt vector (which interrupts are currently active?)
[16]    unused
[15:1]  interrupt mask (if bit is set, the interrupt is recognized)
[0]     enabled
```

8.1.2 Inter-processor interrupt (IPI) registers

The IPI registers are summarized below:

```
---------------------------------
Address         32-bit Register
---------------------------------
BASE + 0x80     IPI intr mask
BASE + 0x84     IPI interrupt value
BASE + 0x88     Upper word of
                  Message to thread 0
BASE + 0x8c     Lower word of
                  Message to thread 0
...  up to thread 7 ....
BASE + 0xc0     Upper word of
                  Message to thread 7
BASE + 0xc4     Lower word of
                  Message to thread 7
BASE + 0xc8     Lock
---------------------------------
```

In order to generate an IPI interrupt from thread I to thread J, the following sequence is to be followed by thread I:

```
0. disable interrupts
1. acquire lock (via swapa/ldstuba)
2. update core J IPI  message registers
3. clear ir_mask bit for destination core.
4. set ir-bit for destination core.
5. sets ir_mask bit for destination core.
6. release lock.
7. enable interrupts.
```

The interrupt handler in thread J will respond with the sequence:

```
0. disable interrupts
1. acquire lock (via swapa/ldstuba).
2. read message registers, execute ISR.
3. writes message registers.
4. clear IPI interrupt bit for core J.
5. release lock.
6. enables interrupts.
```

8.1.3 TIC state machines

Each TIC follows a state machine with the following states.

```
IRC_DISABLED
IRC_ENABLED
IRC_INTERRUPTING
```

The state transitions are described by the following state machine:

- On reset:

  ```
  state := IRC\_DISABLED
  control_reg := 0
  ```

- When state is IRC_DISABLED: if there is a write to the control register with [0] = '1' then move to IRC_ENABLED. Else stay here.
- When state is IRC_ENABLED: if an unmasked interrupt is active, then move to IRC_INTERUPTING else if there is a write to control register with bit 0 = 0, then move to IRC_DISABLED.
- When state is IRC_INTERRUPTING: encode the active unmasked interrupt to the interrupt level for the corresponding thread (INTR_LEVEL_¡core-id¿_¡thread-id¿). If there is a write to control register with bit 0 = 0, then move to IRC_DISABLED.

Thus, each TIC goes through the state sequence

```
IRC_DISABLED -> IRC_ENABLED -> IRC_INTERRUPTING -> IRC_DISABLED
```

8.1.4 VHDL

```
entity afb_multicore_interrupt_controller
Described in         GlueModules.vhdl
compile into library GlueModules
```

8.2 The default count-down timer

The timer is a memory mapped device with a CPU-side interface that consists of request/addr/write-data input-pipes and a read-data ouptut pipe.

It generates a single output interrupt which connects to the interrupt controller.

The timer has a single memory mapped 32-bit control register which is currently mapped to virtual memoy 0xffff3100. The control regiser fields are:

```
31:1    max-timer-count.
0       enable bit
          when set, the timer does a count-down from the
          max-timer-count and when the count reaches 0,
          asserts the interrupt.
```

Its behaviour is as follows:

- Starts in disabled state. When control-reg-bit 0 is set, it moves to the enabled state, and starts the count-down-timer.

- Stays in the enabled state until the count reaches 0, after which the timer moves to the interrupting state in which the timer interrupt is asserted (held) to the system constant TIMER_IRL which has value 0xa.

- The timer stays in the interrupting state until the CPU either explicitly enables the timer (moves to enabled state) or disables the timer (moves to disabled state).

- If at any point, the CPU enables/disables the timer (by doing a register write), the timer will immediately move to either the enabled/disabled state and perform the appropriate actions.

8.2.1 VHDL

```
entity afb_timer
Described in         GlueModules.vhdl
compile into library GlueModules
```

8.3 The simple serial device

The serial device is a memory mapped I/O device that allows for exchange of bytes between the CPU and an external I/O device such as a UART. It is assigned an interrupt level of 12.

The serial device has three internal registers

- an 8-bit rx_register into which data read from the input pipe is stored (this is mapped to address 0xffff3220).
- an 8-bit tx_register into which data from the CPU is written (mapped to address 0xffff3210).
- a 5-bit control/status register (mapped to address 0xffff3200) which has the following bit-fields

```
4: rx-register-full.
    if set indicates that the rx-register has data that needs
    to be read.  The rx-register will not be updated until
    data has been read by the CPU.

 NOTE: writes to the control register will not modify
 this bit.  The bit is set when data is written
 into the rx-register.  The bit is cleared when
 the rx-register is read.
3: tx-register-full.
    if set indicates that the tx-register has been updated by
    the CPU but has not yet been transmitted.
 NOTE: writes to the control register will not modify
 this bit.  The bit is set when data is written
 into the tx-register.  The bit is cleared when
 the tx-register is transmitted.

2: rx-interrupt-enable.
    if set indicates that an interrupt should be raised whenever
      the rx-register has been updated.
1: rx-enable
    if set (by the CPU) indicates that the receive function is
    active.
0: tx-enable
    if set (by the CPU) indicates that the transmit function is
    active.
```

- Two registers for setting the baud rate.

```
register                address
-------------------------------------------------
```

```
baud-limit-register      0xffff320c
baud-frequency-register  0xffff3210
-------------------------------------------------
```

These registers are set by the

```
__ajit_serial_set_baudrate__
```

function provided as part of the AJIT access routines (see Chapter 9).

Internally, its behaviour is modeled by two state machines, the rx and tx state machines.

The rx state machine starts in idle. If rx-enable is set, it moves to an enabled state. In the enabled state, it looks to read from the input pipe and if data is read, moves to the received state (and if rx-interrupt-enable is set, raises an interrupt). It stays in the received state until the rx-register is read by the CPU, at which point it moves to either disabled/enabled state as indicated by the rx-enable bit.

The tx state machine starts in idle. If tx-enable is set, it moves to an enabled state. In the enabled state, it waits for a tx-register write from the CPU, and when this happens, it moves to the transmit state, and sets tx-full to 1. In the transmit state, it writes the tx-register to an internal pipe to a transmit daemon, and goes to a transmit-done-wait state. In this state, when it receives an OK from the transmit daemon, it clears rx-full = 0, and moves to either disabled/enabled state as indicated by the tx-enable bit.

8.3.1 Using the serial device: RX

To use the serial device on the RX side, the cpu must enable the rx function. The serial device will then attempt to read a byte into the rx-buffer. If the cpu has also enabled the rx-interrupt function, an interrupt is generated when the byte is read into the rx-buffer. The rx-full flag is also asserted. When the CPU reads the rx-buffer, the rx-full flag is deasserted and the rx state goes back to enable. Thus, the CPU should use the RX in the following ways

```
rx-enable ->
    poll-rx-full ->
        read-rx-register ->
          disable (or leave enabled).

rx-enable ->
    wait-intr ->
      read-rx-register ->
          disable (or leave enabled).
```

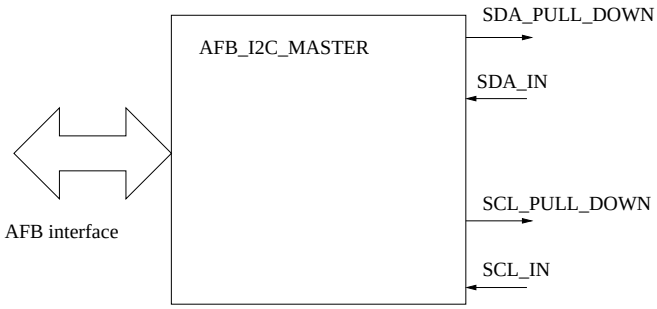

Figure 8.2: AFB SPI Flash controller

8.3.2 Using the serial device: TX

To use the serial device on the TX side, the cpu must enable the tx function. When the cpu writes to the tx-buffer, the serial device sets tx-full, and will attempt to write the byte out, and when it succeeds, will clear tx-full. The cpu should monitor tx-full and if it is empty, disable the tx function.

```
tx-enable ->
   write-tx-buf ->
       poll-tx-full ->
             disable.
```

8.3.3 Restrictions on accesses to the serial registers

The serial control register **must** be accessed by single word load/stores (**st** and **ld** instructions). However, the Tx and Rx registers **must** be accessed by byte load/store instructions (**ldub** and **stub**).

8.3.4 VHDL

```
entity afb_serial_adapter
Described in         GlueModules.vhdl
compile into library GlueModules
```

8.4 AFB I2C master controller

This provides an interface from the AFB bus (AJIT peripheral bus) to an I2C master. The block diagram is shown in Figure 8.2.

There are three addressable registers in the I2C master.

```
name        address  contents
            offset
register-0  0x0      [31:0] = clock divider count.
```

```
register-1  0x4      [31:0] = command register
register-2  0x8      [31:0] = response register.
```

All registers must be accessed as 32-bit wide word accesses.

The clock divider count can be programmed by the user. If the clock frequency to the I2C master is X, and the desired I2C frequency is Y, then the clock divide count should be set to

$$(X/Y)/4$$

The command register has the following format:

```
[31]    rwbar
[30:29] write-mask
   [30] if set send register address
   [29] if set send write data.
[28:22] unused
[22:16] device address
[15:8]  register address in device.
[7:0]   write data byte.
```

The write-mask bits determine which bytes are sent to the I2C bus. For an I2C device register access, bits [30:29] are set 11. But it is possible to send just the register address by using 10, and just the write data byte by using 01.

The response register has the format

```
response format: [31:11] unused
                 [10]  ready
                 [9]   ack-error
                 [8]   busy
                 [7:0] read data
```

The user of the I2C master is expected to poll the response to determine if the master is ready or busy, and reads back the value received from the I2C slave device from the read data field. An error bit is also provided.

8.4.1 VHDL

```
entity              afb_i2c_master
Described in file   simpleI2Clib.vhdl
Library             simpleI2CLib
```

8.5 Self Tuning UART

A self tuning UART will use an initial received byte (value 0x80) sent from the external calibrator to estimate the clock ticks per baud.

Subsequent to the initial byte, the UART will function normally.

The VHDL entity for the self-tuning UART is mySelfTuningUart, compiled into library myUartLib. After reset, the calibrator will be required to transmit 0x80 to the UART.

The ports of the self-tuning UART are

```
entity mySelfTuningUart is
  port (
      -- rising edge of clock is used, reset is active high.
      clk, reset: in std_logic;
      -- Rx, Tx
      UART_RX: in std_logic_vector(0 downto 0);
      UART_TX: out std_logic_vector(0 downto 0);
      -- Fifo interface to user logic (user logic writes
      --    into FIFO TX_to_CONSOLE)
      TX_to_CONSOLE_pipe_write_data: in std_logic_vector(7 downto 0);
      TX_to_CONSOLE_pipe_write_req:  in std_logic_vector(0 downto 0);
      TX_to_CONSOLE_pipe_write_ack:  out std_logic_vector(0 downto 0);
      -- Fifo interface to user logic (user logic reads
      --    from FIFO CONSOLE_to_RX)
      CONSOLE_to_RX_pipe_read_data : out std_logic_vector(7 downto 0);
      CONSOLE_to_RX_pipe_read_req :  in std_logic_vector(0 downto 0);
      CONSOLE_to_RX_pipe_read_ack :  out std_logic_vector(0 downto 0));
  end entity mySelfTuningUart;
```

The calibration sequence is as follows:

- Reset the UART
- Write 0x80 into the UART to calibrate, using the calibrate routine (see below).
- Wait 1ms.
- Start normal use of the UART.

A utility routine calibrateUart is provided to do the calibration. From the terminal, type

```
calibrateUart /dev/ttyUSBx
```

where USBx is the tty corresponding to your UART.

8.5.1 VHDL

```
entity mySelfTuningUart
Described in          myUartLib.vhdl
Compiled into library myUartLib
```

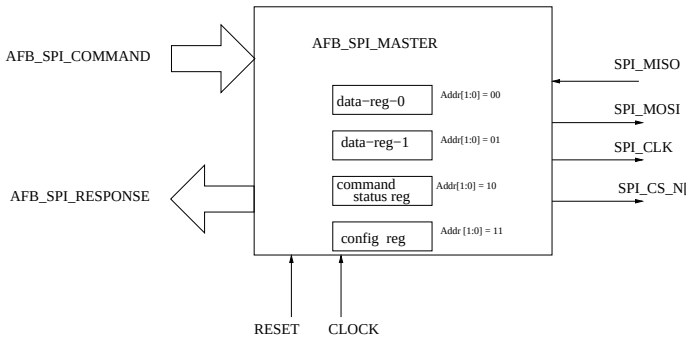

Figure 8.3: AFB SPI master

8.6 AFB SPI Master

The AFB SPI master provides SPI master functionality with an AFB interface. Using the AFB bus, it is possible to address four internal registers to achieve the SPI master function. The structure of the AFB SPI master is shown in Figure 8.3. The SPI-master implementation is based on an implementation by Hans Huebner (Copyright 2009-2010) from opencores.org.

The internal registers are addressed using the bottom two bits of the AFB address as follows:

```
addr      register
00        data-reg-0[7:0]  (data low)
01        data-reg-1[15:8] (data_high)
10        Command/status
11        Config
```

The two registers data-reg-1 and data-reg-0 form a shift register of length 16. Serial data out from the SPI master (MOSI) is tapped out from either at half-word (16-bit) or 12/8/4 bit points along the shift register. Incoming bits go into data-reg-0[0].

When a byte is written to the command/status register, it is interpreted as follows (bit-fields):

```
[7:6] unused
[5:3] SPI slave address
[2]   IRQ-en
[1]   Deselect after transfer
[0]   start-transfer
```

When a byte is read from the command/status register, it provides the following information

```
[7:1]  unused
[0]    busy
```

The busy bit is set to indicate that a transfer is ongoing.

When a byte is written to the config register, it is interpreted as follows:

```
[5:4] transfer-length
     00=4 bits,  01=8 bits
     10=12 bits, 11=16 bits
[3:0] clock-divide count
     0000=clk/2, 0001=clk/4 etc.
      upto 1111=clk/131072.
```

The SPI-CLOCK is obtained by dividing the system clock by

$$2^{clkdividecount+1}$$

The default is divide by 8. The transfer length coding is

```
00   4-bits
01   8-bits  (default)
10  12-bits
11  16-bits
```

To transmit a byte using the SPI-master, we go through the following sequence:

- Configure the SPI clock, transfer length.
- Wait until the master is free (check busy bit of status register to check).
- Write a byte into data register.
- Write a command to the command register to initiate the transfer.
- The clearing of the busy bit in the status register indicates end of transfer. An interrupt can also be generated.

To receive a byte using the SPI-master.

- Wait until the master is free (not busy).
- Write command to CS to initiate the transfer.
- Wait until master is free.
- Read the data buffer(s).

8.6.1 VHDL details

The details of the VHDL implementation of the AFB SPI master are as follows:

```
entity            afb_spi_master
Described in file GenericCoreAddOnLib.vhdl
library           GenericCodeAddOnLib
```

8.7 AFB SPI Flash controller

This controller provides a simple AFB interface to a serial SPI Flash memory such as the ones on the Kintex 705 FPGA cards from Xilinx (See Xilinx documentation note XAPP586 from www.xilinx.com). The AFB SPI Flash controller provides read-write access to serial (SPI) flash memory. Alternate resources on an FPGA card can be used to write to the flash memory. With this controller, the flash memory can be accessed using the normal AFB request response protocol.

The AFB SPI flash controller will serve the memory request specified on the AFB request using an SPI Flash device. Currently, SPI flash devices from Micron have been validated. The SPI flash device must have a capacity between 128KB (1 Mbit) and 16 KB (128 Mbit).

The WRITE_PROTECT input on the AFB SPI flash controller is used as follows:

- When WRITE_PROTECT is tied to '1', then all writes to the flash controller are ignored.
- When WRITE_PROTECT is tied to '0', the flash controller will serve writes. Further, the first write after reset also triggers an ERASE operation in the flash device, which initializes all storage bits in the flash to '1', ready to be written. This erase operation can take several seconds to complete.

8.7.1 VHDL

The AFB SPI Flash read/write controller VHDL details are listed below:

```
entity afb_flash_rw_controller
described in VHDL file  GlueModules.vhdl
compile into library    GlueModules
```

8.8 Processor core add-ons

There are some useful building blocks which can help you put together a system using the generic AJIT processor core. We describe these add-ons briefly.

8.8.1 GlueModules

The file GlueModules.vhdl contains several important building blocks, summarized below:

```
-----------------------------------------------
GlueModules.vhdl
-----------------------------------------------
```

```
acb_afb_bridge
  ACB to AFB bridge, will convert 64-bit
  ACB bus to 32-bit AFB bus.

acb_fast_mux
  ACB request/response multiplexor.

acb_fast_splitter
  ACB splitter with 0-delay

acb_fast_tap
  ACB fast tap with 0-delay

acb_null_stub
  ACB null device, returns zero response

afb_acb_bridge
  AFB to ACB bridge.

afb_fast_mux
  ACB request/response multiplexor.

afb_fast_splitter
  AFB fast splitter with 0-delay

afb_fast_tap
  AFB fast tap with 0-delay

afb_flash_rw_controller       (read/write)
  flash read-write controller, interfaces to
  AFB bus and to SPI flash with sizes in
  range 128KB to 16MB.

afb_multicore_interrupt_controller
  interrupt controller.

afb_null_stub
  AFB null device, returns zero response

afb_scratch_pad
  AFB compatible 32x32 scratchpad.

afb_serial_adapter
  AFB serial adapter..  Need to connect
  this to UART for complete functionality.
```

```
afb_timer
  AFB timer.

afb_trace_logger
  AFB trace logger.
```

8.8.2 GenericCoreAddOnLib

The file GenericCoreAddOnLib.vhdl contains more useful building blocks.

```
ahblite_controller
  AHB controller.

afb_ahb_bridge
  AFB - AHB bridge

afb_apb_controller
  AFB - APB bridge.

acb_sram_stub
  ACB SRAM

afb_sram_stub
  AFB SRAM

afb_2port_sram_stub
  AFB dual-ported SRAM

afb_splitter
  AFB splitter (slower than the fast splitter
  in GlueModules, deprecated).

ahb_sram_stub
  AHB SRAM

apb_sram_stub
  APB SRAM

afb_spi_bridge
  AFB to SPI bridge

afb_spi_master
  AFB SPI master.

afb_spi_flash_controller  (read-only)
```

```
  flash read-only controller, interfaces to
  AFB bus and to SPI flash with sizes in
  range 128KB to 16MB.

afb_gpio
  32-bit GPIO

afb_mux
  AFB multiplexor
```

Chapter 9

AJIT access routines and support utilities

In order to program systems using the AJIT processor, several utilities are provided. These routines are kept in the AJIT tool-chain area in the following locations:

```
AjitPublicResources
     ajit_access_routines_mt
            include
            src
            asm

     minimal_printf_timer
            include
            src
```

The AJIT access routines consist of hardware access functions, and peripheral access functions. For more information, see

```
AjitPublicResources
     ajit_access_routines_mt
         include/ajit_access_routines.h
```

9.1 Trap handlers and interrupt handlers

In the directory

```
AjitPublicResources
     ajit_access_routines_mt
         asm
```

there are several important assembly routines which are essential for applications. These include

- Trap handlers:

```
trap_handlers_for_rtos.s
   window-overflow/underflow trap-handlers.
```

- Interrupt service routines:

```
generic_isr_single_stack_mt.s
    generic isr which runs on the same
    stack as the interrupted task.

generic_isr_with_separate_stack_mt.s
    generic isr which uses a separate
    stack from the interrupted task.
```

- Mutexes:

```
mutexes.s
    mutex using swap instruction
    and using ldstub instruction.
```

- Software trap handlers:

```
generic_sw_trap_mt.s
    generic software trap handler.
```

- System calls:

```
generic_sys_calls.s
    system-call implementations.
```

Chapter 10

AJIT C reference model and compilation tools

10.1 The AJIT C reference model

The AJIT C reference model is a processor emulator, which models a multi-core multi-threaded AJIT processor, including the execution threads, the memory units and some peripherals. The peripherals that are modeled are: a serial device, timer, interrupt controller, and a scratch-pad memory. During the execution of the model, it is possible to connect a debugger (gdb), set break and watch points and monitor contents of registers/memory in the processor model. It is also possible to define a post-condition on the register and memory contents and to confirm that the values in the registers/memory are consistent with the expected values. This post condition is checked after the processor has reached the error state.

The model is invoked as follows:

```
ajit_C_system_model
```

The principal options for this model are:

```
-m <mmap-file>
   required, specifies memory-map of processor for this test.
[-n <number-of-cores>]
   Optional.

   The number of cores to be modeled.  This number can be between
   1 and 4.  The default is 1.
     (NOTE: In each cpu, asr29 holds the cpu-id which can be 0/1/2/3).

[-u 32/64]
   Optional.
```

```
   If -u 64 is specified, the 64-bit extensions to the Sparc-V8
   ISA are also modeled.  Otherwise (the default), only Sparc-V8
   instructions are decoded and the extensions cause an unimplemented
   instruction trap.

[-g]
   Optional, run the CPU in debug mode.
[-p <gdb-port-number>]
   Required with -g, to specify remote debug port.
[-d]
   Optional, check post-condition.
[-r <results-file>]
   Required with -d, specifies expected register/memory
   values at end of run (post-condition).
[-l <log-file>]
   Required with -d, specifies a log-file of the
   post-condition checks.
[-q <number-of-address-bits>]
   Optional, size of memory is 2**<number-of-address-bits>, default is 32.
[-w <reg-writes-dump>]
   Optional, if specified, a log of all register and memory writes is
   generated (a separate file is generated for each cpu core).
```

As an example, you could invoke

```
ajit_C_system_model -m add_test.mmap -n 4\
          -d -l add_test.log -r add_test.results\
          -w add_test.wtrace
```

This uses add_test.mmap as the memory map file and uses add_test.results as the post-condition. It also generates a write-trace of the executed instructions for each cpu core that is modeled.

Another example:

```
ajit_C_system_model -m add_test.mmap -g -p 8888
```

This loads add_test.mmap and waits to connect to a remote debug session via port 8888.

The typical performance of this model is 150K instructions per second per core.

10.2 Compilation tools

The cross-compilation tools sparc-linux-* are available. These include the C compiler (sparc-linux-gcc), the assembler (sparc-linux-as), the linker (sparc-linux-ld) and other useful GNU tools for porting code to the AJIT processor.

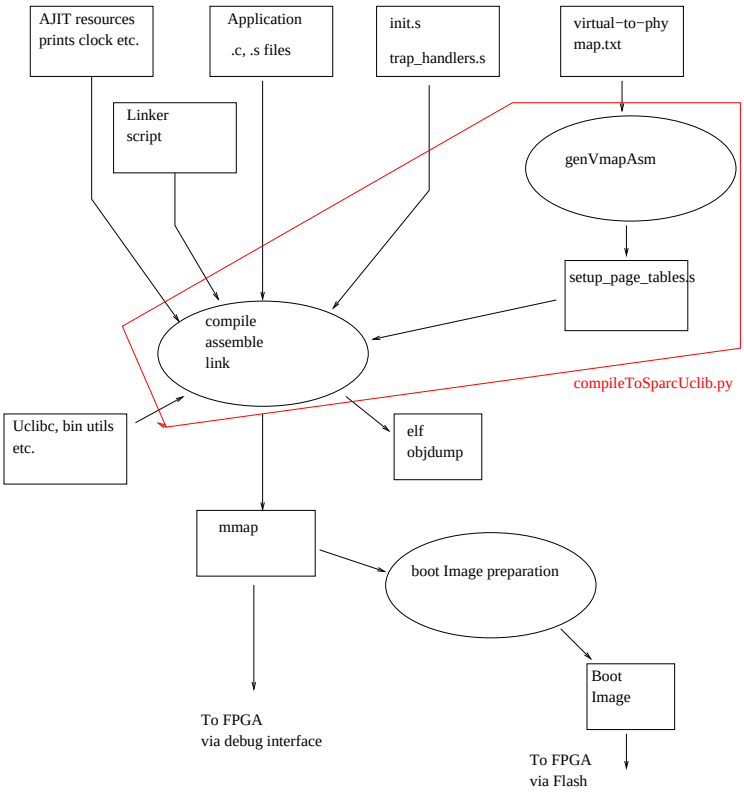

Figure 10.1: Overall compilation flow

The compilation flow is shown in Figure 10.1. The components of this flow are declared in more detail below.

10.2.1 compileToSparcUclibc.py

For compiling bare-metal applications to the AJIT core, the following utility is to be used.

```
compileToSparcUclib.py  ... options ...
```

The script generates elf, hex-dump, object-dump and memory-map files, given the source code that is to be compiled, assembled and linked.

The options are as follows

```
[-h]
   help message and quit.
[-W work-area]
   output directory (default ./)
[-L linker-script]
```

```
   linker directives script file to be used.
-N name of the application..
   If the name is foo, then foo.elf will be generated.
-I dir
   include directory (multiple can be specified)
-C dir
   all *.c files in dir will be compiled.
-c file.c
   file.c will be compiled.
-S dir
   all *.s files in dir will be assembled.
-s file.s
   file.s will be assembled.
-D define-string
   define-string will be passed to C compiler
[-U] uclibc flag
   if specified, the final executable links to
   uclibc
[-g]
   C files will be compiled with -g
[-o 0/1/2/3]
   C files will be compiled with -O0/-O1/-O2/-O3
(-F <compiler-option>)*
   You can pass a compiler option.. for example if
   you specify -F frename-registers, then the option
   "-frename-registers" will be passed to the compiler.
```

Thus, it is possible to specify the assembly files, C files, include directories, library directories, defines, linker script, virtual to physical memory map and use these to produce a final executable.

10.2.2 makeLinkerScript.py

You can generate a basic linker script (passed to the compileToSparc.py using the -L option) using the makeLinkerScript.py script. This is used as follows:

```
makeLinkerScript.py options
```

The options are as follows

```
(-h)? (print-help message)
(-t <text-section-address>)
       address to start text-segment, default= 0x0
(-d <data-section-address>)
       address to start data-segment, default 0x1000
(-o <output-file-name>)
```

For example, to map data sections to 0x40000000 and higher, and to start the text section at 0x0, use:

```
makeLinkerScript.py -t 0x0 -d 0x40000000 -o customLinkerScript.lnk
```

This produces the following linker script.

```
/* Linker script generated for AJIT standalone application */
/* command: makeLinkerScript.py -t 0x0 -d 0x40000000 -o customLinkerScript.lnk */
ENTRY (_start)
__DYNAMIC = 0;
SECTIONS
{
    . = 0x0;
    .text ALIGN(4) : {
     KEEP(*(.text.ajitstart))
     KEEP(*(.text.pagetablesetup))
     KEEP(*(.text.traphandlers))
     KEEP(*(.text.traptablebase))
     *(.text) *(.text.*) }
    . = 0x40000000;
    .rodata ALIGN(4) : { *(.rodata) *(.rodata.*) }
    .data ALIGN(4) : { *(.data) *(.data.*) *(.bss)}
}
```

10.2.3 genVmapAsm

This utility is very useful when you wish to map sparse virtual memory to a compact physical memory. The utility is invoked as

```
genVmapAsm v_to_p_mapping.txt  setup_page_tables.s
```

where v_to_p_mapping.txt contains virtual to physical page mappings and setup_page_tables.s is the assembly file which generates the page tables in memory for use during execution of the program.

Each line of the mapping file has the form

```
context-id virtual-addr phy-addr page-level [cacheable acc]
```

The cacheable and acc have default values 1 and 3 respectively. The acc value is to be specified as per the MMU specification in the Sparc-V8 ISA manual.

An example of a v_to_p_mapping.txt file is

```
0x0 0x0 0x0 0x1
0x0 0x40000000 0x80000 0x2
0x0 0xfffff000 0xff000 0x3 1 3
```

which specifies the following mappings (all for context 0): a level-1 page (16MB) from virtual address 0x0 to physical address 0x0, a level-2 page (256kB) from virtual address 0x40000000 to physical addres 0x80000, and level-3 page (4kB) from virtual address 0xfffff000 to physical address 0xff000 (with cacheable=1, and acc=3).

10.3 Setting up the runtime environment on bare-metal: the init.s file

Before running a program on the processor (in a bare-metal scenario), we need to set up the stack and the virtual to physical mappings. A sample initialization file which does this is shown below

```
.global _start;
_start:
  set -256, %sp
  clr %fp

  ! note: wim  is setup assuming that
  !  we start from window 7 (below).  Window 0
  !  is marked invalid..
  !
  ! you will need to supply overflow/underflow
  ! trap handlers.
  set 0x1, %l0
  wr %l0, 0x0, %wim

  ! trap table.
  set  trap_table_base, %l0
  wr  %l0, 0x0, %tbr

  ! set up virtual -> physical map.
  ! (the page_table_setup code  is
  !   in setup_page_tables.s)
  call page_table_setup
  nop

  ! update the context-table-pointer.
  ! (the set_context_table_pointer code
  !   is in setup_page_tables.s)
  call set_context_table_pointer
  nop

  ! enable traps.
  set 0x10E7, %l0
  wr %l0, %psr

  ! enable mmu.
  set 0x1, %o0
  sta %o0, [%g0] 0x4

  ! the main program.. GO
```

```
call main
nop
```

Chapter 11

Preparing a FLASH image for booting an application

We describe the procedure to be used in order to prepare a FLASH image for booting an application.

We assume that

- FLASH memory is mapped to physical address range 0x0 to 0x3fffffff
- SRAM is mapped to physical address range 0x40000000 to 0xfffeffff
- Physical devices are mapped to address range 0xffff0000 to 0xffffffff

Typically, we will be given an application, which is to be loaded into SRAM by executing a program which resides on FLASH memory.

11.1 Compiling the application

We first compile and link the application you want to boot. The overall flow is illustrated in Figure 11.1.

For the assumptions shown above, we instruct the linker to place the text section at address 0x40000000, and the data section at 0x80000000. This can be done using the linker script shown below:

```
ENTRY(_start)
__DYNAMIC = 0;
SECTIONS
{
. = 0x40000000;
.text ALIGN(8) : {
KEEP(*(.text.ajitstart))
     *(.text)
     *(.text.*)
```

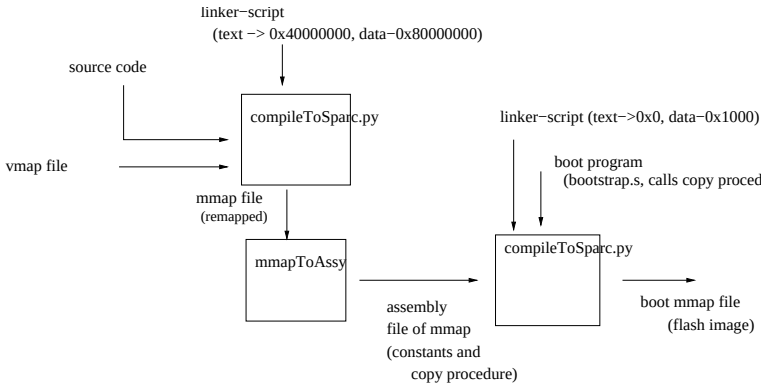

Figure 11.1: Preparing a boot flash image

```
     KEEP(*(.text.pagetablesetup))
     KEEP(*(.text.traphandlers))
     KEEP(*(.text.traptablebase))
}
. = 0x80000000;
.rodata ALIGN(8) : { *(.rodata) *(.rodata.*) }
.data ALIGN(8)   : { *(.data) *(.data.*) *(.bss) }
}
```

We may further use a vmap file for the compilation.

```
! I cacheable.
0x0 0x40000000 0x40000000 0x2  0x1 0x3
! D cacheable.
0x0 0x80000000 0x40020000 0x2  0x1 0x3
! Stack cacheable..
0x0 0xffffe000 0x400fe000 0x3  0x1 0x3
! I/O non-cacheable
0x0 0xffff0000 0xffff0000 0x3  0x0 0x3
```

This uses 20 bits of physical memory address space to map the virtual address from 0x40000000 upwards.

The compilation produces an .mmap.remapped file.

11.2 Prepare the boot image

The mmapToAssyU64 utility is used to convert the application's .mmap.remapped file to an assembly program.

```
mmapToAssyU64 rpn.mmap.remapped assy.s 0x40000000 0x20000
```

This produces an assembly program which writes the application code image starting from address 0x40000000 into a data block of size 0x20000 (Note: this means that the application code image size must be at most 0x20000 bytes. In case of overflow, the routine will complain).

The assy.s file which is produced is then linked with a bootstrap bootstrap.s file (in directory tools/flash_image/asm). The bootstrap.s file contains the following:

```
.section .text.ajitstart
.global _start;
_start:
set -256, %sp
clr %fp

set 0x1, %l0 ! window 0 is marked invalid...  we start at window 7
wr %l0, 0x0, %wim !

! enable traps.
set 0x10E7, %l0
wr %l0, %psr

! copy to sram..
call copy_program_image
nop

! jump to code.
call jump_to_code
nop

ta 0

!
! This is a small subroutine which copies
! memory bytes from one region to another.
!
! g3 contains starting source address, g4 contains
! number of bytes (must be > 0) to be copied,
! g5 contains the starting destination addres.
!
.global _copy_segment;
_copy_segment:
         ldub [%g3], %l3
stub %l3, [%g5]
add  %g3,1, %g3
add  %g5,1, %g5
         subcc %g4, 1, %g4
```

```
bnz _copy_segment
nop
retl
nop
```

After linking the assy.s and bootstrap.s files, using a linker script that maps text to location 0x0, we get a bootstrap mmap file (lets say it is bootstrap.mmap). This needs to be written into the flash memory. In order to prepare a bin file of the mmap, use

```
mmapToBin 20 bootstrap.mmap bootstrap.bin
```

The bootstrap.bin file is copied to FLASH (0x0 to 0x..) and can be booted from address 0x0.

Chapter 12

The AJIT FPGA based prototyping platforms

We have two FPGA based prototyping platforms currently available for evaluating the AJIT processor and for developing software for it.

- A KC705 based single board computer with flash memory and DRAM, which can be used to boot Linux and for programs which need large amounts of memory.
- A VC709 based bare platform with 4MB of SRAM and a large amount of resources available for building SOC prototypes.

12.1 KC705 board-based FPGA prototype of AJIT Single-board-computer

The KC705 based AJIT FPGA prototyping platform allows the use of the AJIT core in a single board computer setup with

- 16MB of FLASH memory.
- 128MB of DRAM.
- Serial UART for debug.
- Serial UART as a terminal driver.
- Integrated interrupt controller and timer.

The Linux 3.16.1 kernel can be booted on this platform. The internal structure of the platform on the KC705 board from Xilinx is shown in Figure 12.1.

Some points to note about this platform are

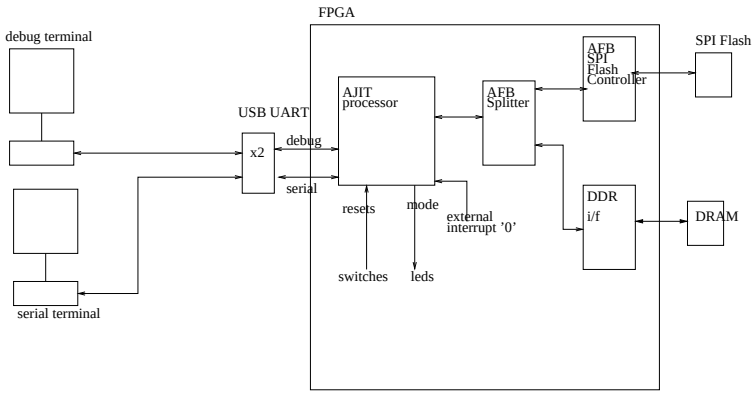

Figure 12.1: A standalone AJIT prototype platform

- The 64-bit bus (The AJIT core bus) from the processor core is split into two busses, one for controlling a flash memory device (16MB) and the other for connecting to the Xilinx DRAM controller (configured to control 128MB of DRAM).
- Two serial UARTs are provided: one for the debug connection, and one for a terminal.

The AFB splitter, and AFB SPI Flash Interface are building blocks that are included in the generic AJIT core distribution (see Chapter 8). The AFB splitter is used to divide the address space into a low space (for Flash, 0x0 to 0x3fffffff) and a high space (for DRAM, 0x40000000 to 0xfffeffff). The AFB SPI Flash interface is used to control the SPI Flash memory device.

12.1.1 Using the KC705 platform

As indicated in Figure 12.1, the setup has the following:

- a debug terminal, which can be used to control and observe the prototype system using the ajit_debug_monitor_mt (ADM) utility. More details about the use of the ADM are provided in Chapter 10.
- a serial terminal: To provide terminal I/O to programs running on the prototype platform.
- The prototype FPGA card, which needs to be programmed with a bit-file which implements the processor and the peripherals inside the FPGA.

One case use the prototype in two ways:

- Load the program into high memory, and run it directly from there.
- Create a boot image to flash, and boot the image from flash.

These two options are described in more detail in Chapter 10.

12.1.2 Writing programs for the KC705 platform

From a software perspective, the platform consists of the processor core, peripherals and main memory. The memory map of the system is

```
FLASH
          0x0 to 0xffffff
DRAM
          0x40000000 to 0x407ffffff

Interrupt-controller
          0xffff3000 to 0xffff301c

Timer (count-down)
         0xffff3100  to 0xffff311c

Serial device
         0xffff3204  to 0xffff3210

Scratch pad
         0xffff2c00  to 0xffff2ffc
```

A typical program would first enable the peripherals of interest and then jump to a main program. Details about using the peripherals are described in Chapter 8. Examples of test programs are given in the AJIT tool-chain repository (with and without the CORTOS2 environment).

We recommend that you use the CORTOS2 environment, which provides a simple abstract view of the platform.

12.2 VC709 board-based FPGA prototype of AJIT Single-board-computer

The VC709 based AJIT FPGA prototyping platform allows the use of the AJIT core in a bare system on chip setup with:

- 4MB of SRAM (single cycle block RAM on the FPGA).
- Serial UART for debug.
- Serial UART as a terminal driver.
- Integrated interrupt controller and timer.

The internal structure of the system on the VC709 board from Xilinx is shown in Figure 12.2.

Some points to note about this platform are

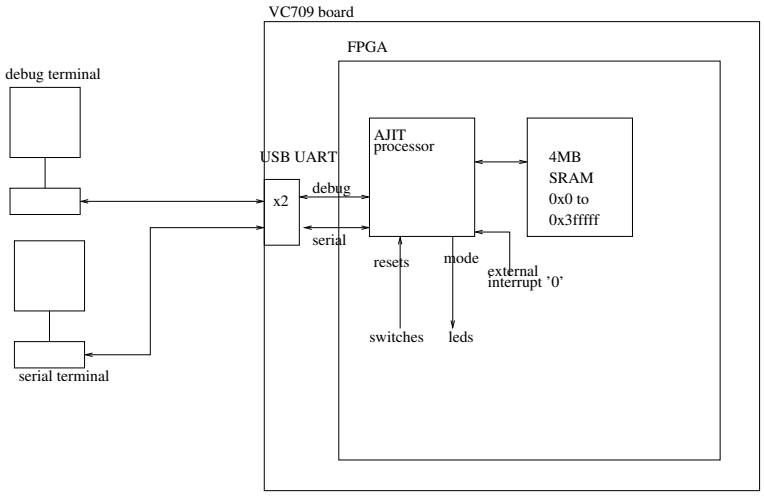

Figure 12.2: A standalone AJIT prototype platform on the VC709 card

- The 64-bit bus from the processor core is connected to a 4MB SRAM implemented using FPGA block RAM.
- Two serial UARTs are provided: one for the debug connection, and one for a terminal.

12.2.1 Writing programs for the VC709 platform

From a software perspective, the platform consists of the processor core, peripherals and main memory. The memory map of the system is

```
SRAM
          0x0 to 0x3fffff

Interrupt-controller
          0xffff3000 to 0xffff301c

Timer (count-down)
         0xffff3100  to 0xffff311c

Serial device
         0xffff3204  to 0xffff3210

Scratch pad
         0xffff2c00  to 0xffff2ffc
```

A typical program would first enable the peripherals of interest and then jump to a main program. Details about using the peripherals are described in Chapter 8. Examples of test programs are given in the AJIT tool-chain repository (with and without the CORTOS2 environment).

We recommend that you use the CORTOS2 environment, which provides a simple abstract view of the platform.

12.2.2 Setup for using the VC709 platform

As indicated in Figure 12.2, the setup to use the platform consists of the following components:

- a debug terminal, which can be used to control and observe the prototype system using the ajit_debug_monitor_mt (ADM) utility. More details about the use of the ADM are provided in Chapter 10.
- a serial terminal: To provide terminal I/O to programs running on the prototype platform.
- The prototype FPGA card, which needs to be programmed with a bit-file which implements the processor and the peripherals inside the FPGA.

One case use the prototype to run a program by first loading the program into memory starting from address 0x0, and then running the program on the processor. This options is described in more detail in Chapter 10.

12.2.3 Differences between KC705 and VC709 platform

The differences between the two platforms are as follows:

- The VC709 has a 4MB memory region mapped from address 0x0 onwards.
- The KC705 has a 16MB FLASH region starting from 0x0, and a 128MB RAM region starting from address 0x40000000. The KC705 implementation includes a FLASH controller integrated with the processor.

You will need to take this into account when mapping an application to the VC709.

Chapter 13

The AJIT debug monitor for multi-core/multi-thread processors

The AJIT CPU includes a debug support unit (DSU) which allows full controllability and observability of the CPU state. Using the DSU, the developer can

- Observe all registers and memory locations.
- Control all registers and memory locations.
- Set hardware break points.
- Set hardware watch points.

The AJIT debug monitor can be used to connect to the DSU, and serve as the shell for a debugging session.

13.1 On chip debug scheme

Let us take the example of the 2x2 processor configuration. Within this processor, we have two cores, each of which has two threads. Thus there are four threads, which can be identified by a pair of numbers core-id, thread-id.

Each thread has a debug support unit interface which consists of command and response pipes. The processor has a debug multiplexor which manages the connection with each of the CPU's in the processor. The multiplexed debug connection connects to a host computer using USB-UART device.

On the host side, a debug server with one thread per AJIT CPU being debugged is started. At the same time, client debug sessions (one for each AJIT

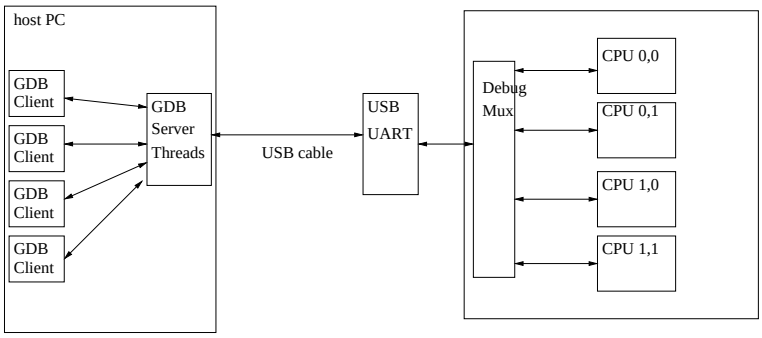

Figure 13.1: AJIT Multi-CPU debug setup

CPU) are started. Once the client connects to the server, the control of that CPU is managed by the GDB remote client in the usual way.

This setup is shown in Figure 13.1. Using this debug setup, it is possible to concurrently debug all the AJIT CPU threads that are active inside the processor.

13.2 The AJIT debug monitor utility for multiple threads

The AJIT debug monitor utility uses the hardware debug interface unit in the AJIT processor in order to control execution in a system which uses the AJIT processor core.

The setup of the debug monitor utility is shown below.

```
-------------------------------------------------------------------
 ajit_debug_monitor_mt   <--- uart --->   AJIT multi-threaded
  runs on PC        FPGA prototype
-------------------------------------------------------------------
```

The AJIT debug monitor utility is invoked at the command line

```
    ajit_debug_monitor_mt -u <tty-dev> [-n <ncores>] [-t <nthreads-per-core>]

    The options are
      -u tty-dev required, when the processor core is
        mapped to an FPGA board, with a serial UART (tty-dev)
        being used as the debug connection.
      -n <ncores> , optional, specify number of cores
                        (default = 1, maximum = 4)
      -n <nthreads_per_core> , optional, specify number of threads
in each core (default = 1, maximum = 2)
```

The AJIT debug monitor provides the user with a command shell for an interpreter. The command prompt is

```
ajit[0:0]>
```

The two numbers [0 : 0] in the prompt specify the core and thread id which is currently being controlled/observed in the monitor. To change the core and thread being monitored, we can at any time use

```
ajit[0:0]>  t 0 1
```

This will change the core and thread id if possible (note the -n and -t options above). The prompt now changes to

```
ajit[0:1]>
```

For each prompt, the user can supply a command. The commands are classified as follows.

```
Reset/mode control/observe.
Initial PC/NPC/PSR control/observe.
State register control/observe.
Integer general purpose register observe/control.
Floating-point general purpose register observe/control.
Memory control/observe.
Load memory map file.
Run script file.
Start/stop GDB server.
Help, quit, log.
```

13.2.1 Reset/mode control/observe

Using the interpreter, it is possible to control the reset values applied to the core, as well as to observe the mode it is running in.

```
w rst <reset-val>
```

Write reset-val to the 4-bit CPU reset input to the AJIT core. The value must be an integer value, specified either using the hex (0x...) or decimal format. For example:

```
w rst 0x1
```

puts the processor in reset mode, and

```
w rst 0x0
```

brings it out of reset.

The reset values constitute a 8-bit field, of which only the bottom 5 bits are used, as follows:

```
[4]  :  if set, the  CPU will generate a debug
        break on any trap/interrupt.
[3]  :  if set, the processor will produce a logging
        trace (this is reserved, and may not be
        supported in your platform).
[2]  :  if set, the CPU will run in single-step
        mode.
[1]  :  if set, the CPU connects to debugger before executing
        the first instruction, after reset is released
[0]  :  if set, initialize the CPU, Caches, MMU. This
                    must be cleared to start running code.
```

To read the CPU mode, we use

```
r mode
```

This returns the CPU mode. This is a 2-bit value, and should have one of the following values

```
value    meaning
 0x0     uninitialized mode
 0x9     in reset mode.
 0x2     in normal run mode.
 0x3     in error mode.
```

Note: the processor core will stay in reset mode as long as the least significant bit of the controlled rst value is 1.

13.2.2 Initial PC/NPC/PSR control/observe

We can set the initial value of PC, NPC, PSR so that the processor starts (post-reset-release) from the desired instruction and status register state.

```
w ipc <init-pc-val>
```

Set the initial PC at which the processor core starts when it comes out of reset. For e.g.,

```
w ipc 0x40000000
```

means when the CPU comes out of reset, the initial PC is 0x40000000.

```
r ipc
```

returns the value of initial PC.

```
w inpc <init-npc-val>
```

Sets the initial NPC at which the processor core starts when it comes out of reset. For e.g.,

```
w inpc 0x40000004
```

means when the CPU comes out of reset, the initial NPC is 0x40000004.

```
r inpc
```

returns the value of initial NPC.

```
w ipsr <init-psr-val>
```

Sets the initial PSR at which the processor core starts when it comes out of reset. For e.g.,

```
w ipsr 0x10c0
```

means the when the CPU comes out of reset, the initial PSR is 0x10c0.

```
r ipsr
```

returns the value of initial PSR.

13.2.3 State register control/observe

It is possible to control and observe the current value of the state registers using the interpreter.

```
w psr/wim/tbr/y <hex-value>
```

Set the current value of one of PSR/WIM/TBR/Y registers to hex-value. e.g.

```
w wim 0x1
```

sets the WIM value in the processor core to 0x1.

```
r psr/wim/tbr/y
```

Read the current value of PSR/WIM/TBR/Y registers. e.g.

```
r tbr
```

returns the current value of the TBR.

```
w asr <asr-id> <asr-value>
```

Sets the current value of one of ASR[asr-id] to asr-value. e.g.

```
w asr 0x1  0xff
```

sets the value of ASR[0x1] to 0xff.

```
r asr <asr-id>
```

Read the current value of ASR[asr-id] register. e.g.

```
r asr 0x1
```

returns the current value of ASR[0x1].

13.2.4 Integer general purpose register control/observe

It is possible to observe control/observe the general purpose register values using the interpreter.

```
w iureg <reg-id> <reg-value>
```

Set the current value of one of integer general purpose register [reg-id] to reg-value e.g.

```
w iureg 0x7  0xff
```

sets the value of R[0x7] to 0xff.

```
r iureg <reg-id>
```

Read the current value of the integer unit R[reg-id] register e.g.

```
r iureg 0xf
```

returns the current value of R[0xf].

13.2.5 Floating point general purpose register control/observe

It is possible to observe control/observe the floating point register values using the interpreter.

```
w fpreg <reg-id> <reg-value>
```

Set the current value of one of floating point general purpose register [reg-id] to reg-value e.g.

```
w fpreg 0x7  0x10000000
```

sets the value of F[0x7] to 0x10000000.

```
r fpreg <reg-id>
```

Read the current value of the floating point unit F[reg-id] register. e.g.

```
r fpreg 0xf
```

returns the current value of F[0xf].

13.2.6 Memory control/observe

It is possible to observe control/observe any memory location using the interpreter (including the various supported ASI's). Observation and control is possible at the word (32-bit) level only.

```
w mem <asi-value> <addr-value> <write-value>
```

Write 32-bit write-value to address addr-value in memory space defined by asi asi-value. Address is force aligned to 32-bit access by setting bottom two bits 0. e.g.

```
w mem 0xa  0x0 0xffffffff
```

sets the value of mem[0x0] with asi=0xa to 0xffffffff.

```
r mem <asi-value> <addr-value>
```

Read the current 32-bit value of the memory (identified by asi) location addr-value. e.g.

```
r mem 0xa 0x0
```

returns the current value of mem[0x0], with memory defined by asi=asi-value.

13.2.7 Load memory map file

It is possible to load a memory map into system memory by using the interprter. The memory map file consists of byte-address, byte-value pairs, both specified in hex format.

```
m <mmap-file>
```

loads the memory map in mmap-file to the processor memory.

13.2.8 Execute script file

A list of debug monitor commands can be listed in a file, and the file can be executed as a script. Comment lines in the script start with a "!" character.

```
s <script-file>
```

executes the AJIT debug monitor commands listed in the script-file.

```
e.g. script file
! Comment: reset the core..
w rst 0x1
! load mmap file.
m mmap.txt
! bring core out of reset.
w rst 0x0
! read mode
r mode
```

13.2.9 Start/stop GDB server

The AJIT debug monitor can start a GDB debug server on the host machine to allow GDB to remotely connect to the current thread and control a debugging session. Note that the GDB connection and the AJIT debug monitor use the same connection to communicate with the host. Thus, when GDB is active, the debug monitor should be quiet.

```
g  start <port-id>
```

Start the GDB server to listen on port port-id.

```
g  stop
```

Stop the GDB server.

13.2.10 Help, quit, log

These are useful commands to get basic online help, to quit the interpreter, and to generate a command log for re-use.

```
q
```

quit the monitor.

```
h
```

print a help message.

```
l <log-file>
```

generate a log of successful commands into log-file.

13.2.11 Setting up the environment

We assume that the processor model is set up and ready to use. The processor+system is mapped to an FPGA card, with the debug interface being mapped to UART (card can be standalone, FPGA must be programmed). The reset switch inputs to the processor on the FPGA card must be 0.

Further the AJIT tool chain must be set up on the host machine.

13.2.12 Typical use cycle

Suppose we are monitoring a one core, four thread processor. In typical use, one follows the sequence (script file shown below):

```
! put the thread (0,0) in reset
w rst 0x1
! set initial values for pc/npc/psr in thread (0,0).
w ipc 0x0
w inpc 0x4
```

```
w ipsr 0x10c0
! put the thread (0,1) in reset
t 0 1
w rst 0x1
! set the initial values for pc/npc/psr in thread (0,1)
w ipc 0x0
w inpc 0x4
w ipsr 0x10c0
! download the memory map (m <mmap-file>)
m ADD.mmap
! release the reset for the current thread 0 1
w rst 0x0
! change thread to 0 0
t 0 0
! release the reset for the current thread 0 0
w rst 0x0
```

Now both threads are running and you can monitor their modes.

```
r mode
        t 0 1
r mode
        .. etc ..
! quit / go back to reset mode and try again.
q
```

You can start a GDB server through the debug monitor. The debug monitor and GDB server use the same connection to the processor. Thus, once you start a GDB server, you cannot resume normal debug monitor activity.

The following sequence must be followed for a 1-core two thread system.

```
! put thread 0 0 in reset mode
w rst 0x1
! set initial values for pc/npc/psr for thread 0 0
w ipc 0x0
w inpc 0x4
w ipsr 0x10c0
! put thread 0 1 in reset mode.
t 0 1
w rst 0x1
! set initial values for pc/npc/psr for thread 0 1
w ipc 0x0
w inpc 0x4
w ipsr 0x10c0
! download the memory map (m <mmap-file>)
m ADD.mmap
! start GDB server on port 8889 for thread 0 1
```

```
g start 8889
! change thread to 0 0
t 0 0
! start GDB server on port 8888 for thread 0 0
g start 8888
! bring thread 0 0 out of reset, but in debug-mode
w rst 0x2
! change to thread 0 1
t 0 1
! bring thread 0 1 out of reset, but in debug-mode
w rst 0x2
```

From here on, the debug monitor should be quiet until the debugging session is over. Two GDB clients will be needed to connect to the two GDB servers started above.. .. eventually ...

```
q
```

Note that at the end of the GDB session, you need to quit the debug monitor.

Chapter 14

Performance counters

Each AJIT execution thread can log the following events.

- Execution of an instruction (issued and executed).
- Skipping of an instruction (issued, but not executed).
- L1 Dcache load access.
- L1 Dcache load miss.
- L1 Dcache store access.
- L1 Dcache store miss.
- Branch mispredict.
- Trap.
- L1 Icache access.
- L1 Icache miss.

A performance counter peripheral is available. This allows the programmer to access the counts of these events on a per-thread basis. When used, the connection of this performance counter peripheral is shown in Figure 14.1. The event logging signals from each thread are used to maintain per-thread sets of counters for the event classes listed above. The performance counter peripheral has an AFB interface which can be used to access the performance counters in the peripheral. Currently, the default builds of the AJIT processor which include the performance counters use the address mapping range

```
0xffff4000 - 0xffff40ff    for thread 0,0
0xffff4100 - 0xffff41ff    for thread 0,1
0xffff4200 - 0xffff42ff    for thread 1,0
etc. for threads 1,1  and so on.
```

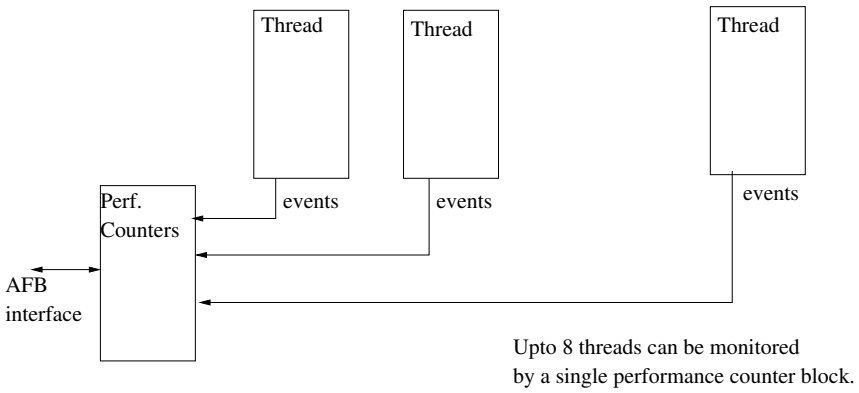

Figure 14.1: Performance counters: connections

and within the per-thread space, the registers holding the performance counters have the addresses (bottom 8-bits):

```
-------------------------------------------
addr   register
-------------------------------------------
0x0    executed_instruction_count[63:32]
0x4    executed_instruction_count[31:0]
0x8    skipped_instruction_count[63:32]
0xc    skipped_instruction_count[31:0]
0x10   load_count[63:32]
0x14   load_count[31:0]
0x18   load_miss_count[63:32]
0x1c   load_miss_count[31:0]
0x20   store_count[63:32]
0x24   store_count[31:0]
0x28   store_miss_count[63:32]
0x2c   store_miss_count[31:0]
0x30   mispredict_count[63:32]
0x34   mispredict_count[31:0]
0x38   trap_count[63:32]
0x3c   trap_count[31:0]
0x40   icache_access_count[63:32]
0x44   icache_access_count[31:0]
0x48   icache_miss_count[63:32]
0x4c   icache_miss_count[31:0]
-------------------------------------------
```

In addition, in each thread, there is a free running counter which counts the number of clock cycles. This counter is accessed by reading the ASR register pair 30,31.

14.1 Software interface

In the AJIT access routines provided as part of the AJIT tool chain, the following data structure can be used to maintain the performance counters for a hardware CPU/Thread.

```
typedef struct __AjitPerThreadPerformanceCounters {
  uint64_t executed_instruction_count;
  uint64_t skipped_instruction_count;
  uint64_t load_count;
  uint64_t load_miss_count;
  uint64_t store_count;
  uint64_t store_miss_count;
  uint64_t stream_mispredict_count;
  uint64_t trap_count;
  uint64_t icache_access_count;
  uint64_t icache_miss_count;

} AjitPerThreadPerformanceCounters;
```

The following interface functions are available to update this data structure.

```
// initialize all performnce counters to 0.
void __ajit_init_thread_performance_counters (int core_id,
          int thread_id,
          AjitPerThreadPerformanceCounters *tpc);

// samples the values of the performance counters.
void __ajit_sample_thread_performance_counters (int core_id,
          int thread_id,
          AjitPerThreadPerformanceCounters *tpc);
```

These can be called when the thread is in either user mode or supervisor mode.

To access the 64-bit cycle count register in the thread, we can use

```
uint64_t __ajit_get_clock_time();
```

Note that this function can be called only when the thread is in supervisor mode.

Bibliography

[1] The SPARC Architecture Manual, https://sparc.org/technical-documents/#V8.

[2] Madhav Desai, "The **Aa** language reference manual," https://github.com/madhavPdesai/ahir.

[3] M. Jacobsen et.al., "RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators," *ACM Transactions on Reconfigurable Technology and Systems,* Vol. 8, No. 4, Article 22, September 2015.