Generic AJIT processor core description and user guide

Madhav P. Desai
Department of Electrical Engineering, IIT Bombay

October 11, 2019

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

Generic 32-bit AJIT processor core overview

This document describes the use of the generic 32-bit AJIT processor core, an implementation of the SPARC-V8 ISA [1]. The processor core has been developed at IIT Bombay over the period 2015-2019. The processor core was designed using the AHIR-v2 toolset [2], which has also been developed at IIT Bombay over the period 2012-2019.

Depending on the licensing agreement with IIT Bombay, the 32-bit AJIT generic processor core platform is distributed with the following components:

Core implementation Distributed either as high-level **Aa** description [2], synthesizable VHDL or an FPGA netlist file (NGC/EDIF).

Core bin-utils C/C++ cross-compiler, uclibc and binary utilities (linker, assembler, debugger, elf-tools).

Compilation scripts Python scripts for cross-compilation and linking.

Core models System emulator in C.

Core documentation ISA description, default device memory and interrupt mapping, I/O interface documentation.

AHIR release A snapshot of the AHIR-v2 toolset, built for x86_64 Linux based systems.

Examples examples high-lighting the features of the processor core.

In the rest of this document, we will describe the generic 32-bit AJIT processor core, and methods to use it both from a programmer's point of view and from a system-on-chip designers point of view.

1.1 Processor core block diagram and interface description

In Figure 1.1, we show the structure of the generic core. The generic core itself is build around a base core (indicated by the blue box). The base core has the following components:

- AJIT processor central processing unit (CPU): implements the SPARC-V8 ISA (Draft IEEE standard 1754-1996). The entire instruction set (except for quad-floating-point instructions) is implemented. Eight register windows are implemeneted. A detailed description of the implementation decisions taken in the CPU is given in Chapter 2.
- Instruction Cache (ICACHE): A 32-kB (64 byte line size), direct mapped, virtually indexed and virtually tagged instruction cache.
- Data Cache (DCACHE): A 32-kB (64 byte line size), direct mapped, virtually indexed and virtually tagged data cache with write-through allocate policy.
- Memory management unit (MMU): implements all required aspects of the SPARC reference MMU.
- A 64-bit AJIT Core Fifo Bus interface.

In the generic core, the base core is packaged with some additional elements, so that the base core can be immediately put to use in building simple systems. These additional elements are:

- A bridge which translates the 64-bit base core requests to a 32-bit protocol called the AJIT FIFO BUS (AFB) protocol.
- Integrated peripherals: essential peripherals are integrated into the generic core itself. These include a count-down timer (max timer count is 32-bits), a serial device and an interrupt controller. Device mappings are described in detail in Chapter 4.
- Debug support unit with UART: The AJIT CPU includes a debug support unit which allows remote debugging as well as internal state monitoring and control. It can be used to connect to a remote GDB session.

The I/Os of the generic core are:

Reset/configuration inputs

- CPU_RESET : in std_logic_vector(0 downto 0); if asserted, processor is put into reset. (this is a soft reset, which can be a delayed version of power-on reset).
- DEBUG_MODE : in std_logic_vector(0 downto 0); if asserted, processor will wait for connection from debugger for remote debug session.

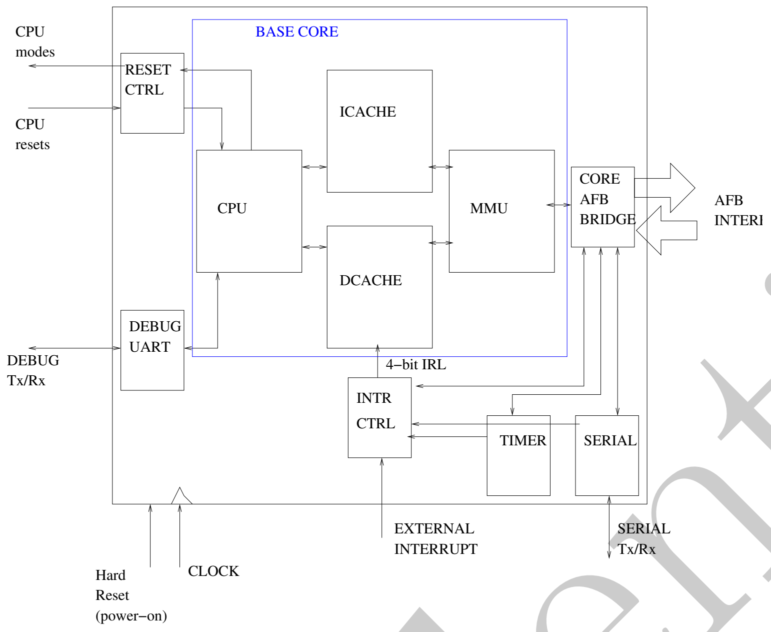


Figure 1.1: The Generic 32-bit AJIT processor core

- SINGLE_STEP_MODE : in std_logic_vector(0 downto 0); if asserted, processor will run a single instruction at a time through the pipeline.

Processor operating mode indicators
- CPU_MODE : out std_logic_vector(1 downto 0);

```
Values:
  "00"  uninitalized
    (after power-on reset)
  "01"  reset mode
    (after CPU\_RESET is asserted)
  "10"  run mode
    (after CPU\_RESET is released)
  "11"  error mode
    processor can be put into error mode
    in two ways
    - trap on trap
    - memory access error.
```

External interrupt (mapped to IRL 13)
- EXTERNAL_INTERRUPT : in std_logic_vector(0 downto 0);

Debug UART connection
- DEBUG_UART_RX : in std_logic_vector(0 downto 0);

- DEBUG_UART_TX : out std_logic_vector(0 downto 0);

Serial UART
- SERIAL_UART_RX : in std_logic_vector(0 downto 0);
- SERIAL_UART_TX : out std_logic_vector(0 downto 0);

AFB Request/response FIFO bus A request and response FIFO interface. The request interface consists of three signals:

- GENERIC_CORE_AFB_REQUEST_pipe_read_data : out std_logic_vector(73 downto 0);
- GENERIC_CORE_AFB_REQUEST_pipe_read_req : in std_logic_vector(0 downto 0);
- GENERIC_CORE_AFB_REQUEST_pipe_read_ack : out std_logic_vector(0 downto 0);

The response interface consists of three signals:

- GENERIC_CORE_AFB_RESPONSE_pipe_write_data : in std_logic_vector(32 downto 0);
- GENERIC_CORE_AFB_RESPONSE_pipe_write_req : in std_logic_vector(0 downto 0);
- GENERIC_CORE_AFB_RESPONSE_pipe_write_ack : out std_logic_vector(0 downto 0);

Details of these signals are given in Section 1.2.

Clock,reset The core works on a single clock (positive edge triggered clk) and has a power-on reset which is active high.

- clk, reset: in std_logic

1.2 AJIT 32-bit FIFO Bus Interface Protocol

The AJIT FIFO BUS (referred to henceforth as AFBx36x32 or AFB for short) interface is a FIFO-based request/response interface which forms the primary interface between the AJIT core and the external world. The AJIT core generates requests using a request FIFO interface and expects responses using a response FIFO interface.

1.2.1 Request FIFO interface

This interface consists of two protocol signals and a 74-bit request word.
The protocol ports are

```
bus_request_read_req: in  std_logic_vector(0 downto 0);
bus_request_read_ack: out std_logic_vector(0 downto 0)
```

The environment asserts the read_req when it is able to accept data. The AJIT core asserts read_ack when it has data to presnt to the environment. When both req/ack are asserted on a rising clock-edge, the data is exchanged.

Note that the ack may be asserted independently of the req. That is, the ack may be viewed as data-available, and the req may be viewed as data-accept.

The request data is a 74-bit word, with the following bit fields

```
bit 73:  lock bit (see below)
bit 72:  read/write-bar
         If this bit is '1', it is a read.
bits 71:68: byte-mask
         A 4-bit byte-mask which indicates the bytes to be
         written (1000 means the most-significant byte is written)
bits 67:32: physical address is 36 bits wide.
bits 31:0: write-data.
```

The lock-bit indicates that the AJIT processor wishes to lock the memory bus starting from this request. The memory bus is locked for other accesses until a subsequent request from the AJIT processor is made with the lock-bit low.

1.2.2 AFB Response

This interface consists of two protocol signals and a 33-bit response word. The protocol ports are

```
bus_request_write_req: in  std_logic_vector(0 downto 0);
bus_request_write_ack: out std_logic_vector(0 downto 0)
```

The environment asserts the write_ack when it has response data to write to the AJIT core. The AJIT core asserts write_ack when it is ready to accept data from the environment. When both req/ack are asserted on a rising clock-edge, the data is exchanged.

Note that the ack may be asserted independently of the req. That is, the ack may be viewed as data-accept, and the req may be viewed as data-available.

The response data is a 33-bit word, with the following bit fields

```
bit 32:  error-bit
           Set if there was an error (what the error is
           depends on the implementation).  It is ok to
           always leave this bit as '0'.
bits 31:0:  read-data.
```

1.3 AJIT processor base core internal structure

The internal structure of the AJIT processor base core is shown in Figure 1.2.

The CPU contains

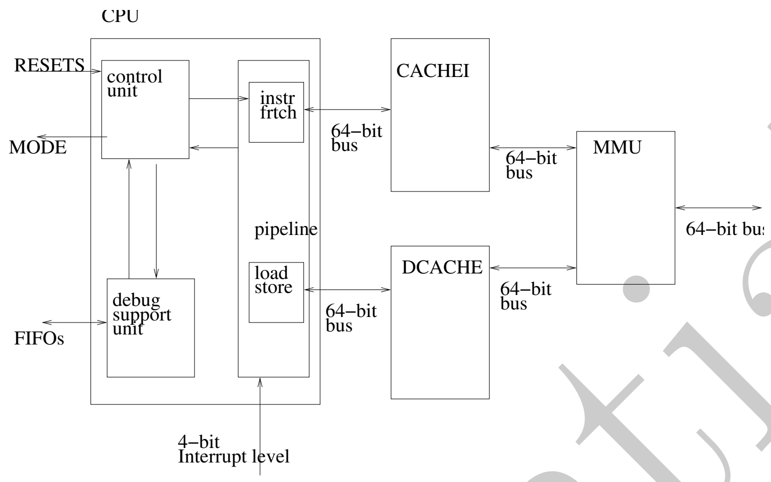


Figure 1.2: 32-bit AJIT processor base core internals

- 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 10 cycles).

The caches are virtually indexed, virtually tagged and can be accessed in two clock cycles (for a hit). The Cache line size is 64 bytes, and the miss penalty is 60 cycles. The MMU implements the SPARC SRMMU standard, with a 4-level page table supporting pages of size 4KB, 256KB, 16MB and 4GB.

In Chapter 2, we provide details on the implementation choices for the AJIT processor core.

1.4 Performance of the generic AJIT core

The performance of the generic AJIT core mapped to FPGA at 100 MHz is summarized below.

- 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.

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 60 cycles on misses.

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.
```

Industry standard benchmark performance is summarized below:

Benchmark	Performance
Dhrystone	2.41 DMIPS/Mhz (optimized) 414K DMIPS iterations/sec at 100 Mhz. (gcc)
Coremark	1.77 Coremarks/Mhz 177 Coremarks at 100 MHz. (gcc)
Whetstone	48 MWIPS at 100 MHz. (with gcc + glibc)

1.5 AJIT processor core FPGA platforms

Two kinds of FPGA platforms are provided. In the first kind, a PCIE interface is used to access the core's peripherals for validation of the core. We refer to this as a tethered PCIE platform. This platform is described in Chapter 11.

In the second kind, the core is mapped to a standalone FPGA card, and we can use peripherals such as UARTs to connect a debugger and serial device to the FPGA platform. We will indicate how to construct such a platform in Chapter 12.

1.6 Summary

We have introduced the AJIT processor core in this Chapter. The AJIT processor base core is a full implementation of the SPARC-V8 ISA, and includes instruction and data caches, a memory management unit and a hardware debug support unit. The base core also provides a 64-bit wide interface to external memory.

The generic AJIT processor core is built on top of the base core. It includes a few additional peripherals and bridges the 64-bit core system bus to a 32-bit AFB. Thus, a basic standalone prototype with UART interfaces can be easily put together using the generic AJIT processor core.

Chapter 2

AJIT processor implementation of the SPARC V8 ISA

The AJIT processor is a complete implementation of the SPARC V8 ISA. Specific decisions made in the implementation are listed below. All design decisions are consistent with the SPARC V8 ISA specification.

2.1 Exceptions

In Figure 2.1, we list the exceptions are generated and handled precisely by the AJIT processor core. Corresponding to an exception, the processor generates a 32-bit trap-vector. We also list the indices of the trap-vector which correspond to the distinct exceptions.

2.2 Trap vectoring

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, $TT\_MASK = 0xfffff00f$, and TRAP_ID is the trap identifier. For the traps listed above, the identifiers are shown in Figure 2.2.

2.3 Instruction-wise implementation decisions

The instructions in the SPARC V8 ISA which are implemented fall into the following classes: data-transfer instructions (listed in Figure 2.3), miscellaneous

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

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).

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

2.4 ASR mappings

ASR[31] and ASR[30] implement a free running 64-bit counter which runs at the processor clock. Writes to these registers are ignored. All other ASR's are implemented as R/W registers without any assigned mappings.

2.5 MMU implementation decisions

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
  64-entry Level-3 TLB
```

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

The following required ASI's are 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

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

of the caches. Such a capability is useful in evaluating a system which excludes the MMU entirely.

```
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

Chapter 3

AJIT processor core memory access paths

We describe the memory access path-ways in the AJIT processor core, when connected to external memory using a 32-bit bus.

The internal structure of the AJIT processor core is shown in Figure 3.1. All the connections between the components of interest correspond to 64-bit data accesses, except for the 32-bit AFB bus connection. The primary components of interest are

- The IFETCH unit: generates instruction fetch requests. Each instruction fetch actually requests a pair of instructions which fit in a double word (64-bits). On average, the IFETCH will issue one instruction fetch request every two clock cycles (except when the PC and NPC instructions do not occupy a single double word).
- The LOADSTORE unit: generates double word loads/stores. The re-

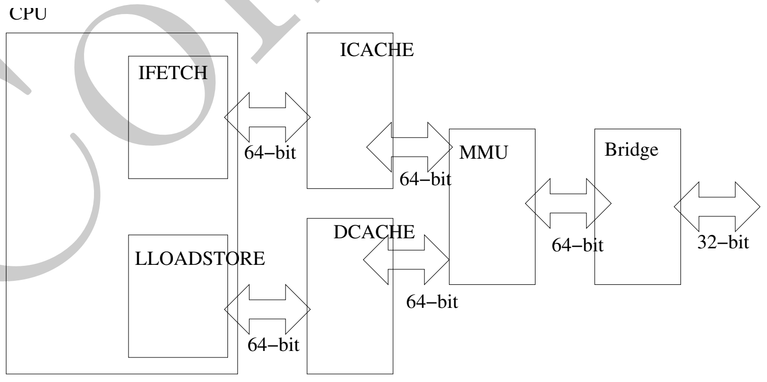


Figure 3.1: The memory paths in the AJIT processor core

quests are always for a 64-bit double word, with an 8-bit byte mask. At the 64-32 bridge (in this case AFB, bridge), a 64-bit request is translated to either one or two 32-bit requests, depending on the byte-mask.

- The ICACHE: receives instruction fetch requests. If there is a hit, the instruction pair is returned immediately. If there is a miss, the request is forwarded to the MMU. If the MMU is enabled, then the MMU will do an address translation, and return a cache line (8x64-bit double words), from which the instruction pair is extracted and returned to the CPU.
- The DCACHE: receives 64-bit load/store requests (with a byte-mask).
  - For loads: If there is a hit, the double word is returned immediately. If there is a miss, the request is passed on to the MMU. If the MMU is enabled, a 8x64-bit cache line is fetched into the DCACHE out of which the relevant double word is extracted and returned to the CPU. Note that each 64-bit access to fill the cache line is implemented as two 32-bit accesses at the AFB bus. If the byte-mask indicates a 32-bit access, then only a single access is performed on the AFB bus.
  - For stores: If there is a hit, the double word is written into the DCACHE and also forwarded to the MMU (write-through policy), to be written into the memory via the AFB interface. Depending on the byte mask, this may cause either one or two 32-bit accesses on the AFB bus. If there is a miss, the request is forwarded to the MMU. If the MMU is enabled, then the double word is written to the memory, and then a 8x64-bit cache line is fetched from the memory and returned to the DCACHE.
- The MMU: receives accesses from either ICACHE or DCACHE. If enabled, the virtual address is translated to the physical address (note: this translation may generate memory accesses on the AFB bus, in case there is a TLB miss). The translated physical address is then used to access the memory. Depending on the type of access from ICACHE/DCACHE, the MMU can perform several actions:
  - ICACHE hit: MMU is not involved.
  - ICACHE miss: If the MMU is enabled, translate virtual address and read the entire cache line (requested double word first) from AFB bus (memory) and return to ICACHE. If the MMU is disabled, read only the requested double word from memory and return it to the ICACHE.
  - DCACHE read hit: MMU is not involved.
  - DCACHE read miss: Same behaviour as in ICACHE miss.
  - DCACHE write hit: If MMU is enabled After translation, MMU writes double word (with byte-mask) to AFB bus.

  - DCACHE write miss: If the MMU is enabled, translate virtual address, and write double word to AFB bus (memory). Then fetch the cache line from memory (8x64-bit accesses) and return the line to DCACHE. If the MMU is disabled, just write the double word to memory.
  - MMU bypass load/stores Certain load/store accesses can be marked as MMU bypass accesses. These are treated as misses by the DCACHE and handled by the MMU as if the MMU is disabled.

- The (64 to 32) Bridge: Accesses generated from the MMU can be byte-wide, half-word-wide, word-wide or double-word-wide, as determined by an 8-bit byte-mask, a 36-bit address and a 64-bit read/write word. Thus, a single access from the 64-bit side can be translated to up to two requests on the 32-bit side.

3.1 Detailed description of the access paths for the different situations, while the MMU is enabled

The possible memory accesses (when the MMU is enabled) can be classified into eleven possible paths as listed below:

Sr.no.	Access-type	Cache	MMU
1.	IFETCH	Hit	-
2.	IFETCH	Miss	Translate, Cacheable
3.	IFETCH	Miss	Translate, Non-cacheable
4.	LOAD	Hit	-
5.	LOAD	Miss	Translate, Cacheable
6.	LOAD	Miss	Translate, Non-cacheable
7.	LOAD	Bypass	Load
8.	STORE	Hit	Translate, Write-through
9.	STORE	Miss	Translate, Cacheable
10.	STORE	Miss	Translate, Non-cacheable
11.	STORE	Bypass	Write-through

These cases are detailed in the following subsections.

```
CPU instr-pair fetch to ICACHE
  ICACHE hit, lookup
Return Instr-pair to CPU
```

Figure 3.2: ICACHE hit path

```
CPU instr-pair fetch to ICACHE
  ICACHE miss, send to MMU
    MMU translate, cacheable
      Request 8x64 to bridge
         Request 16x32 to Memory
         Response 16x32 from Memory
      Response 8x64 to bridge
    Response 8x64 to MMU
  Update line in ICACHE
Return instr-pair to CPU
```

Figure 3.3: ICACHE miss path when MMU translation is cacheable

3.1.1 Case 1: ICACHE hit

In this case, an instruction-pair is returned from the ICACHE itself, without involving the MMU. See Figure 3.2.

3.1.2 Case 2: ICACHE miss, MMU lookup is cacheable

In this case, there is an ICACHE miss, and the MMU is involved in a translation process. The MMU returns a cacheable translation. In this case, the entire line (8x64-bit accesses or 16x32-bit accesses) is fetched from memory to ICACHE (requested word is fetched first). See Figure 3.3.

3.1.3 Case 3: ICACHE miss, MMU lookup is non-cacheable

In this case, there is an ICACHE miss, and the MMU is involved in a translation process. The MMU returns a non-cacheable translation. In this case, a single instruction pair is fetched into the ICACHE (but not saved by the ICACHE) and returned to the CPU. See Figure 3.4.

3.1.4 Case 4: DCACHE load hit

In this case, there is a load hit, and the requested data item (upto one double word) is returned from the DCACHE itself, without involving the MMU. See Figure 3.5.

```
CPU instr-pair fetch to ICACHE
  ICACHE miss, send to MMU
    MMU translate, non-cacheable
      Request 1x64 to bridge
         Request 2x32 to Memory
         Response 2x32 from Memory
      Response 1x64 to bridge
    Response 1x64 to MMU
  Pass 1x64 through ICACHE
Return instr-pair to CPU
```

Figure 3.4: ICACHE miss path when MMU translation is non-cacheable

```
CPU loadstore fetch to DCACHE
  DCACHE hit, lookup
Return requested item to CPU loadstore
```

Figure 3.5: DCACHE load hit path

3.1.5 Case 5: DCACHE load miss, MMU lookup is cacheable

In this case, there is a DCACHE load miss, and the MMU is involved in a translation process. The MMU returns a cacheable translation. In this case, a cache line (8x64-bit or 16x32-bit) is fetched from memory into DCACHE (requested word is fetched first). See Figure 3.6.

```
CPU loadstore load to DCACHE
  DCACHE miss, send to MMU
    MMU translate, cacheable
      Request 8x64 to bridge
         Request 16x32 to Memory
         Response 16x32 from Memory
      Response 8x64 to bridge
    Response 8x64 to MMU
  Update line in DCACHE
Return requested item to CPU loadstore
```

Figure 3.6: DCACHE load miss path when MMU translation is cacheable

```
CPU loadstore load to DCACHE
  DCACHE miss, send to MMU
    MMU translate, noncacheable
      Request data-item to bridge
         Request data-item(s) to Memory
           (note can be 1 or 2 accesses)
         Response data-item(s) from Memory
      Response data-item to bridge
    Response data-item to MMU
  Pass through data-item in DCACHE
Return data item to CPU loadstore
```

Figure 3.7: DCACHE load miss path when MMU translation is non-cacheable

```
CPU loadstore fetch to DCACHE
  DCACHE bypass item
    MMU bypass item
       Request item to bridge
          Request item(s) from memory
            (can be multiple)
          Response item(s) from memory
       Response item to bridge
    Response bypass through MMU
  Response  bypass through DCACHE
Return requested item to CPU loadstore
```

Figure 3.8: DCACHE load bypass

3.1.6 Case 6: DCACHE load miss, MMU lookup is non-cacheable

In this case, there is a DCACHE load miss, and the MMU is involved in a translation process. The MMU returns a non-cacheable translation. In this case, only the requested data item is fetched from memory into DCACHE. The item is not saved in the cache and is returned to the CPU. See Figure 3.7.

3.1.7 Case 7: DCACHE load bypass, MMU is bypassed

In this case, the DCACHE lookup and MMU translation are skipped, and only the requested data item is returned to the CPU. See Figure 3.8.

```
CPU loadstore store to DCACHE
  DCACHE hit, lookup, write
  (DCACHE returns completion to CPU)
    MMU request translate write-through
     (note: no translation error trap)
      Request store to bridge
        Request stores(s) to memory
        Response store(s) from memory
      Response store to bridge
```

Figure 3.9: DCACHE store hit path

3.1.8 Case 8: DCACHE store hit, MMU is used for write-through

In this case, there is a DCACHE store hit, and the data is written into the DCACHE and written through to main memory via the MMU. The MMU does address translation. Note that in this case, the DCACHE offloads a write to the MMU without waiting for a response from the MMU. See Figure 3.9.

3.1.9 Case 9: DCACHE store miss, MMU write-through, lookup is cacheable

In this case, there is a DCACHE store miss (data is written to the DCACHE), and the MMU is involved in a translation process, and does the write through to memory. The MMU returns a cacheable translation. In this case, the entire missed line is fetched from memory to DCACHE (8x62-bit or 16x32-bit). See Figure 3.10.

3.1.10 Case 10: DCACHE store miss, MMU write-through, lookup is non-cacheable

In this case, there is a DCACHE store miss (data is written to the DCACHE), and the MMU is involved in a translation process, and does the write through to memory. The MMU returns a non-cacheable translation. In this case, only the requested data item is read from the memory into the DCACHE, where it is not saved and is returned to the CPU. See Figure 3.11.

3.1.11 Case 11: DCACHE store bypass, MMU write-through, is bypassed

In this case, the DCACHE lookup and MMU translation are skipped, and only the requested data item is returned to the CPU. See Figure 3.12.

```
CPU loadstore store to DCACHE
  DCACHE miss, write item to MMU
    MMU translate, cacheable
      Write data-item to bridge
         Write data-item(s) to Memory
           (note can be 1 or 2 accesses)
         Write data-item(s) response from Memory
      Response ok to bridge
    Response  ok to MMU
  Write response OK to DCACHE
  DCACHE fetch line from MMU
    MMU translate (TLB hit) ok
      Request 8x64 to bridge
        Request 16x32 to Memory
        Response 16x32 to bridge
      Response 8x64 to MMU
   Response to DCACHE, update line
 Return completion to CPU
```

Figure 3.10: DCACHE store miss path when MMU translation is cacheable

```
CPU loadstore store to DCACHE
  DCACHE miss, write item to MMU
    MMU translate, non-cacheable
      Write data-item to bridge
         Write data-item(s) to Memory
           (note can be 1 or 2 accesses)
         Write data-item(s) response from Memory
      Response ok to bridge
    Response  ok to MMU
  Write response OK to DCACHE
Return completion to CPU
```

Figure 3.11: DCACHE store miss path when MMU translation is non-cacheable

```
CPU loadstore store to DCACHE
  DCACHE bypass to MMU
  (Sends response to CPU, no wait on MMU)
    MMU bypass
      Write data-item to bridge
         Write data-item(s) to Memory
           (note can be 1 or 2 accesses)
         Write data-item(s) response from Memory
      Response ok to bridge
    Response  ok to MMU
```

Figure 3.12: DCACHE store bypass

3.2 Memory access paths when the MMU is disabled

If the MMU is disabled, there are two possible behaviours:

- bit 8 of the MMU control register is '0': In this case, every access through the MMU is marked as non-cacheable and all accesses are as shown in the previous section. This is the default behaviour of a Sparc-V8 system.
- bit 8 of the MMU control register is '1': In this case, every access through the MMU is marked cacheable, and accesses are treated as shown in the previous section. This behaviour allows us to configure the processor core to act like an MMU-less system with caching. Be careful, there is no memory protection whatsoever.

3.3 Memory mapped I/O

If there are memory mapped I/O devices, then the memory locations to which these are mapped must be marked as non-cacheable. This can be done in two ways.

- Accesses to the memory mapped I/O locations should be done using a special ASI (bypass). This mechanism can be used by using the alternate space access routines provided in the the ajit access routines code utilities.
- The memory pages in which the I/O devices are mapped should be marked as non-cacheable in the virtual to physical tables of the MMU. For bare-metal applications, you can define a virtual to physical table and use the genVmapAsm utility (see Chapter 6) to generate assembly code that sets up the virtual to physical table in memory. After this is done, you can simply use pointers to access the memory mapped I/O.

3.4 Summary

A single instruction fetch or load/store generated from the CPU can cause multiple transactions on a 32-bit system bus. We have summarized the possible cases in this chapter.

Chapter 4

Device mappings

The generic AJIT core comes with two integrated devices:

- A count-down timer.
- A serial device.

These devices have fixed memory mappings and interrupt levels assigned to them.

In addition, the core includes an interrupt controller which manages these two devices and an external interrupt. The EXTERNAL_INTERRUPT on the generic AJIT processor core is mapped to interrupt level 13.

4.1 The default count-down timer peripheral for the generic AJIT core

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.

4.2 The default serial device in the generic AJIT processor core

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.

It has a CPU-side interface which looks like a memory interface, and an environment-side interface which has two byte-wide pipes/FIFOs.

- CONSOLE_to_SERIAL_RX: an input pipe on which data is received.
- SERIAL_TX_to_CONSOLE an output pipe on which data is transmitted.

In the generic AJIT core, these FIFOs are terminated on a serial UART.

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.
```

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.

4.2.1 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).
```

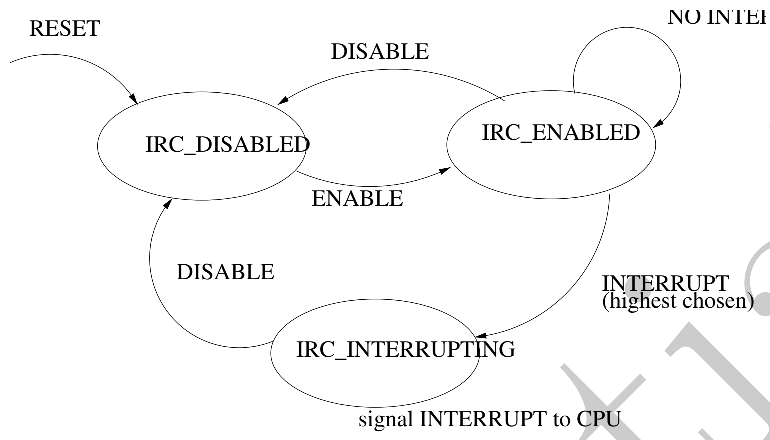


Figure 4.1: The default interrupt controller state machine

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

4.2.2 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.
```

4.3 Generic AJIT core: default interrupt controller (IRC)

The generic AJIT core comes with a default interrupt controller. The interrupt control multiplexes three interrupt sources, one from the timer (IRL=10), one from the serial device (IRL=12) and one from the external interrupt (IRL=13).

The IRC has a single 32-bit control register (mapped to virtual address 0xffff3000), in which bit 0 is the enable bit. The IRC is controlled by the state machine shown in Figure 4.1.

When the CPU sees an interrupt, it jumps to the interrupt service routine corresponding to the interrupt level forwarded by the IRC. The CPU needs to disable the IRC, service the interrupt so that the interrupt device lowers its interrupt, and then enable the IRC again in order to register the next interrupt.

4.4 Device register mappings

In the generic AJIT core, the in-built device memory mappings are as follows:

```
ADDR_INTERRUPT_CONTROLLER_CONTROL_REGISTER      0xFFFF3000
ADDR_TIMER_CONTROL_REGISTER                     0xFFFF3100
ADDR_SERIAL_CONTROL_REGISTER                    0xFFFF3200
ADDR_SERIAL_TX_REGISTER                         0xFFFF3210
ADDR_SERIAL_RX_REGISTER                         0xFFFF3220
```

Chapter 5

AJIT platform tools

The AJIT platform tools are as follow:

- Processor models:
  - C reference simulation model: there are two versions, an older one which models the 32-bit processor with mmu and without caches, and a newer one which models the 64-bit extensions, caches, and can model upto 4 processor cores.
  - Pipelined-processor untimed simulation model.
  - FPGA prototype model.
- Compiler/debugger tools
  - GNU tools: cross-compiler (sparc-linux-gcc), assembler (sparc-linux-as), linker (sparc-linux-ld), debugger (sparc-linux-gdb) etc.
  - Compilation scripts and tools: useful compilation scripts for compiling C, assembly and linking.
  - AJIT utility routines: processor access code, including critical trap handlers and processor setup routines.

5.1 Processor models: C reference ISA model

The first processor simulation model is a C reference models, which can be thought of as a processor emulator. This models the instruction set architecture, the Caches, the MMU and the main memory.

`ajit_C_system_model`

This is a reference C model of the processor, together with a 32kB direct mapped write through DCACHE, a 32 kB direct mapped ICACHE, an MMU and a main memory. A serial device, timer and interrupt controller are also modeled. It is possible to load the memory with initial contents. The model resets the processor to start from PC=0x0, and runs until the processor enters error state.

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 principal options for this tool 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.

5.2 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.

The compilation flow is shown in Figure 5.1. The components of this flow are declared in more detail below.

5.2.1 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
```

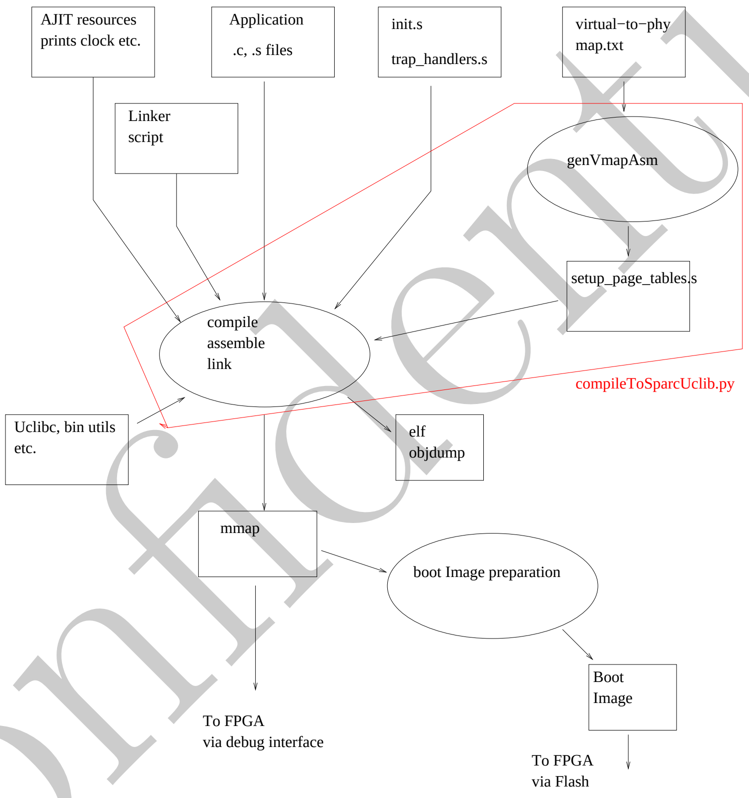


Figure 5.1: Overall compilation flow

```
   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.

5.2.2 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)}
}
```

5.2.3 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).

5.2.4 The init.s file

The setup_page_tables.s file needs to be included as part of the initialization setup in order to set up the page tables before the actual program starts executing. 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 6

The genVmapAsm utility

The genVmapAsm utility is used to generate assembly code which generates a page table in memory. This assembly code can be included in a bare metal program as part of the initialization sequence. Using this scheme it is possible to provide memory protection and I/O memory mapping in a bare-metal program, by using the MMU in the AJIT processor core.

6.1 Synopsys

The genVmapAsm utility is invoked as follows

```
genVmapAsm <vmap-file> <output-assembly-file>

   For example:
     genVmapAsm vmap.txt setup_page_tables.s
```

This uses the virtual to physical mapping described in file vmap.txt and generates assembly code in setup_page_tables.s, which can be used to set up page tables corresponding to this map.

6.2 Format of input file to genVmapAsm

The input file contains a list of page mappings. Each page mapping line has the following format:

```
<context-id> <virtual-page-address>
         <physical-page-address> <page-level>
                   [cacheable-flag]  [acc-code]
(note: the cacheable-flag (default=1)
      and acc-code (default=3) are optional).
```

For example, the following line

```
! context-id  vpage-addr ppage-addr page-level cacheable acc
   0x0         0x0          0x4000  0x3        0x1       0x3
```

means that for context 0, virtual page starting at virtual address 0x0 is mapped to physical page starting at physical address 0x4000, with the page size being 4KB (level 3), with this page being cacheable and allowing read/write/execute access in both supervisor and user mode (acc=3). Note that the base addresses of the physical and virtual pages must be aligned to the page sizes of the corresponding pages.

The access codes (acc) are as per the SPARC V8 reference MMU manual.

6.3 Example

Lets start with a vmap file (suppose this is in file vmap.txt)

```
!
! Line beginning with ! is comment line.
!
! for context 0
!
! 16MB page at 0x0 mapped to 0x0
! cacheable.
0x0 0x0 0x0 0x1
! 256KB page at 0x40000000 mapped
! to 0x80000, cacheable.
0x0 0x40000000 0x80000 0x2
! 256KB page at 0x40040000 mapped
! to 0xc0000, cacheable.
0x0 0x40040000 0xc0000 0x2
!
! 4KB page at 0xffffe000 mapped to
! 0xfe000, cacheable.  We want
! to use this for the stack..
0x0 0xffffe000 0xfe000 0x3
! 4KB page starting at 0xfffff000 is
! mapped to 0xff000  and this is not
! cacheable (can be used for I/O)
0x0 0xfffff000 0xff000 0x3  0x0
```

Now run genVmapAsm

```
genVmapAsm vmap.txt setup_page_tables.s
```

This produces an assembly file called setup_page_tables.s

```
.global page_table_setup
page_table_setup:
```

```
set PAGE_TABLE_BASE, %g1
!PTD: context=0, index=0, level=0, child_p_addr=0x400, p_addr=0x800
! *(PAGE_TABLE_BASE + 0x800) = ptd(PAGE_TABLE_BASE + 0x400)
! make PTD from 0x400
set 0x400, %g4
add %g1, %g4, %g4
srl %g4, 0x4, %g4
or  %g4, 0x1, %g4
! g4 contains PTD
set 0x800, %g5
add %g5, %g1, %g3
st %g4, [%g3]
! g4 stored into [g3]
!PTE: context=0, index=0, level=1,  ppnr=0x0, p_addr=0x400, cacheable=0x1, acc=0x3
! *(PAGE_TABLE_BASE + 0x400) = 0x8e (pte)
set 0x8e, %g2
set 0x400, %g5
add %g5, %g1, %g3
st %g2, [%g3]
!PTD: context=0, index=64, level=1, child_p_addr=0x0, p_addr=0x500
! *(PAGE_TABLE_BASE + 0x500) = ptd(PAGE_TABLE_BASE + 0x0)
! make PTD from 0x0
set 0x0, %g4
add %g1, %g4, %g4
srl %g4, 0x4, %g4
or  %g4, 0x1, %g4
! g4 contains PTD
set 0x500, %g5
add %g5, %g1, %g3
st %g4, [%g3]
! g4 stored into [g3]
!PTE: context=0, index=0, level=2,  ppnr=0x80000, p_addr=0x0, cacheable=0x1, acc=0x3
! *(PAGE_TABLE_BASE + 0x0) = 0x808e (pte)
set 0x808e, %g2
set 0x0, %g5
add %g5, %g1, %g3
st %g2, [%g3]
!PTE: context=0, index=1, level=2,  ppnr=0xc0000, p_addr=0x4, cacheable=0x1, acc=0x3
! *(PAGE_TABLE_BASE + 0x4) = 0xc08e (pte)
set 0xc08e, %g2
set 0x4, %g5
add %g5, %g1, %g3
st %g2, [%g3]
!PTD: context=0, index=255, level=1, child_p_addr=0x2fc, p_addr=0x7fc
! *(PAGE_TABLE_BASE + 0x7fc) = ptd(PAGE_TABLE_BASE + 0x200)
! make PTD from 0x2fc
```

```
   set 0x200, %g4
   add %g1, %g4, %g4
   srl %g4, 0x4, %g4
   or  %g4, 0x1, %g4
   ! g4 contains PTD
   set 0x7fc, %g5
   add %g5, %g1, %g3
   st %g4, [%g3]
   ! g4 stored into [g3]
   !PTD: context=0, index=63, level=2, child_p_addr=0x1f8, p_addr=0x2fc
   ! *(PAGE_TABLE_BASE + 0x2fc) = ptd(PAGE_TABLE_BASE + 0x100)
   ! make PTD from 0x1f8
   set 0x100, %g4
   add %g1, %g4, %g4
   srl %g4, 0x4, %g4
   or  %g4, 0x1, %g4
   ! g4 contains PTD
   set 0x2fc, %g5
   add %g5, %g1, %g3
   st %g4, [%g3]
   ! g4 stored into [g3]
   !PTE: context=0, index=62, level=3,  ppnr=0xfe000, p_addr=0x1f8, cacheable=0x1, acc=
   ! *(PAGE_TABLE_BASE + 0x1f8) = 0xfe8e (pte)
   set 0xfe8e, %g2
   set 0x1f8, %g5
   add %g5, %g1, %g3
   st %g2, [%g3]
   !PTE: context=0, index=63, level=3,  ppnr=0xff000, p_addr=0x1fc, cacheable=0x0, acc=
   ! *(PAGE_TABLE_BASE + 0x1fc) = 0xff0e (pte)
   set 0xff0e, %g2
   set 0x1fc, %g5
   add %g5, %g1, %g3
   st %g2, [%g3]
   retl;
   nop;
! done: page_table_setup
! start: set context-table-pointer = PAGE_TABLE_BASE + 0x800
.global set_context_table_pointer
set_context_table_pointer:
   set PAGE_TABLE_BASE, %g1
   set 0x800, %g5
   add %g5, %g1, %g2
   srl  %g2, 0x4, %g2
   or  %g2, 0x1, %g2
   set 0x100, %g3
   sta %g2, [%g3] 0x4
   retl;
   nop;
! done: set  context-table-pointer
.align 1024
PAGE_TABLE_BASE: .skip 3072
```

6.4 Using the setup_page_tables.s to create a startup routine

Using the setup_page_tables.s file, we can create a startup routine for our program.

```
.global _start;
_start:
  ! frame pointer mapped to 0xffffe000
  mov -4096, %fp
  ! stack pointer is moved down by 256
        sub %fp, -256, %sp

  ! We have 8 windows, and will start at
  ! window 0x7.  Window 0x0 is marked
  ! as invalid.
  set 0x1, %l0
  wr %l0, 0x0, %wim

  !
  ! trap table is at label trap_table_base
  !
  set  trap_table_base, %l0
  wr  %l0, 0x0, %tbr

  !
  ! set up virtual -> physical map.
  !   (this points to code generated by
  !     genVmapAsm).
  call page_table_setup
  nop

  !
  ! set up context table pointer
  !   (this points to code generated by
  !     genVmapAsm).
  call set_context_table_pointer
  nop
```

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

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!  ENABLE MMU                                !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
set 0x1, %o0
sta %o0, [%g0] 0x4

! jump to your main routine.
call main
nop
```

Chapter 7

AJIT platform resources

In order to simplify bare-metal application development, we provide some useful resources.

- access routines for the MMU, peripheral devices.
- a minimal printf and timer library.
- trap handlers with support for window overflow/underflow traps. You can use this a base on which to construct more trap handlers and interrupt handlers etc.

7.1 Ajit access routines

These provide the following routines. You could use these to access processor core and peripheral resources in your applications.

```
// clear all integer registers (across all windows)
inline void __ajit_clear_all_gp_registers__();

// internal timer access.  a 64-bit timer is provided
inline uint32_t __ajit_read_cycle_count_register_high__();
inline uint32_t __ajit_read_cycle_count_register_low__();
inline uint64_t __ajit_get_clock_time();

// set default cacheable bit (see Chapter \ref{ch:IsaImplChoices})
inline void __ajit_set_mmu_default_cacheable_bit__(uint8_t dc_bit);

// store word into mmu register
inline void __ajit_store_word_mmu_reg__(uint32_t value, uint32_t addr);

// store while bypassing the MMU. (this is one way to access peripherals).
inline void __ajit_store_word_mmu_bypass__(uint32_t value, uint32_t addr);
```

```
// load word from mmu register
inline uint32_t __ajit_load_word_mmu_reg__(uint32_t addr);

// load while bypassing mmu (this is one way to access peripherals).
inline uint32_t __ajit_load_word_mmu_bypass__(uint32_t addr);
//
// mmu disable/enable functions.
inline void __ajit_enable_mmu__();
inline void __ajit_disable_mmu__();

// Mmu flush: flush the TLB
inline void __ajit_mmu_flush__(uint32_t value, uint32_t addr);

// Mmu probe: returns the pte corresponding to this address.
inline uint32_t __ajit_mmu_probe__(uint32_t addr);

// make a switch to a particular context
inline void __ajit_mmu_context_switch__(uint32_t context_number);

// page table manipulation.
inline uint32_t __ajit_va_l1_offset__ (uint32_t va);
inline uint32_t __ajit_va_l2_offset__ (uint32_t va);
inline uint32_t __ajit_va_l3_offset__ (uint32_t va);
inline uint32_t __ajit_make_pte__ (uint32_t ppn, uint8_t cacheable,
                          uint8_t modified, uint8_t referenced,  uint8_t acc);
inline uint32_t __ajit_make_ptd__ (uint32_t table_ptr_bits_35_downto_6);

// flush icache
inline void __ajit_flush_icache__(void);

// flush_dcache
inline void __ajit_flush_dcache__(void);


// bypass access timer
inline void __ajit_write_timer_control_register__(uint32_t val);
inline uint32_t __ajit_read_timer_control_register__();

// memory mapped io access timer.
inline void __ajit_write_timer_control_register_via_vmap__(uint32_t val);
inline uint32_t __ajit_read_timer_control_register_via_vmap__();

// bypass methods to access serial device
inline void __ajit_write_serial_control_register__(uint32_t val);
inline uint32_t __ajit_read_serial_control_register__();
inline void __ajit_write_serial_tx_register__(uint32_t val);
inline uint32_t __ajit_read_serial_tx_register__();
inline uint32_t __ajit_read_serial_rx_register__();
int   __ajit_serial_putchar__  (char c);
int   __ajit_serial_getchar__ ();
void  __ajit_serial_puts__ (char* s, uint32_t length);
void  __ajit_serial_gets__ (char* s, uint32_t length);
void __ajit_serial_configure__ (uint8_t enable_tx, uint8_t enable_rx, uint8_t enable_intr);


// memory mapped IO methods to access serial device
inline void __ajit_write_serial_control_register_via_vmap__(uint32_t val);
inline uint32_t __ajit_read_serial_control_register_via_vmap__();
inline void __ajit_write_serial_tx_register_via_vmap__(uint32_t val);
inline uint32_t __ajit_read_serial_tx_register_via_vmap__();
inline uint32_t __ajit_read_serial_rx_register_via_vmap__();
int   __ajit_serial_putchar_via_vmap__  (char c);
int   __ajit_serial_getchar_via_vmap__ ();
void  __ajit_serial_puts_via_vmap__ (char* s, uint32_t length);
void  __ajit_serial_gets_via_vmap__ (char* s, uint32_t length);
void  __ajit_serial_configure_via_vmap__ (uint8_t enable_tx, uint8_t enable_rx, uint8_t enable_in

//  bypass based access to interrupt controller.
inline void __ajit_write_irc_control_register__(uint32_t val);
inline uint32_t __ajit_read_irc_control_register__();

// memory mapped io access to interrupt controller.
inline void __ajit_write_irc_control_register_via_vmap__(uint32_t val);
inline uint32_t __ajit_read_irc_control_register_via_vmap__();


// Trap: generate TICC trap.
//---------------------------------------------------------------------------------------------
inline void __ajit_ta_0__ ();

// *b  = fsqrtd (*a), uses f0,f1,f2,f3 registers..
inline void __ajit_fsqrtd__ (uint32_t a, uint32_t b);
// *b  = fsqrts (*a), uses f0,f1 registers..
inline void __ajit_fsqrts__ (uint32_t a, uint32_t b);
// *b  = fitod (*a), uses f0,f2,f3 registers..
inline void __ajit_fitod__  (uint32_t a, uint32_t b);
// *b  = fitos (*a), uses f0,f1 registers..
inline void __ajit_fitos__  (uint32_t a, uint32_t b);
// *b  = fdtoi (*a), uses f0,f1,f2 registers..
inline void __ajit_fdtoi__  (uint32_t a, uint32_t b);
// *b  = fstoi (*a), uses f0,f1 registers..
```

```
inline void __ajit_fstoi__  (uint32_t a, uint32_t b);
```

7.2 Minimal printf and timer routines

The following print and timer functions are provided for your use.

```
void ajit_serial_init();

// from www.eembc.org (Core mark source)
int ee_printf(const char *fmt, ...);

// return elapsed number of ticks
// (1 tick = 256/clock_frequency seconds).
uint32_t ajit_barebones_clock();
```

7.3 Trap handlers

A basic set of trap handlers is provided. For window overflow and underflow traps, a full trap handler is provided. For other traps, only a place-holder trap handler is available.

Chapter 8

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.

8.1 Compiling the application

We first compile and link the application you want to boot. The overall flow is illustrated in Figure 8.1.

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.*)
```

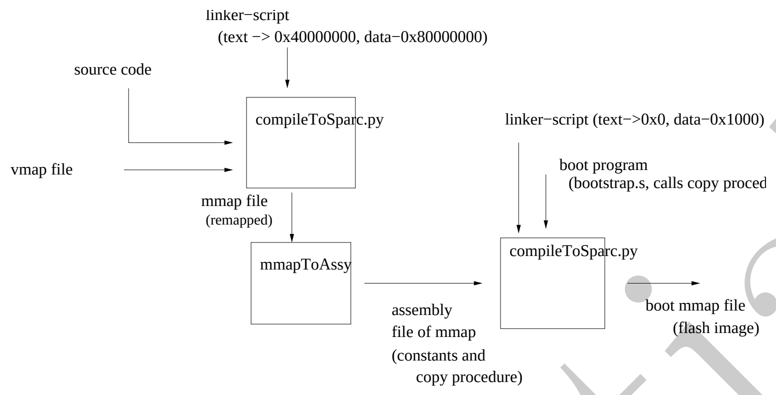


Figure 8.1: Preparing a boot flash image

```
     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.

8.2 Prepare the boot image

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

```
mmapToAssy rpn.mmap.remapped assy.s
```

This produces an assembly program which writes the application code image starting at address 0x40000000 The assy.s file is 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 0x40000000
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, we get an mmap file. This needs to be written into the flash memory. See the example in tools/flash_image/example.

Chapter 9

Remote GDB debugger for the AJIT Processor

The generic AJIT processor core has a debug UART connection which can be used for remote GDB based debugging. In order to use remote debugging, one needs to connect the debug UART to a remote terminal, and start the processor core in DEBUG mode.

In order to debug an application, it must be cross-compiled with the "-g" option. Post compilation , we generate a memory map file and an ELF file. The memory map file can be converted to a flash image or directly downloaded to system memory over the debug UART itself.

9.1 Starting the processor core

We use the application ajit_stand_alone_testbench (see Chapter 5) to start the processor core. Connect the debug UART to a terminal, and in the terminal, type

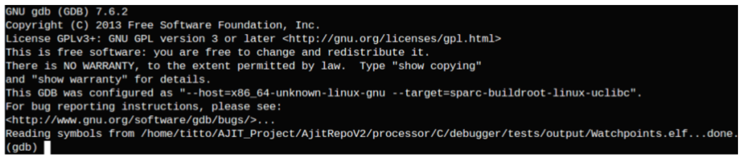
```
ajit_stand_alone_testbench -m <mmap-file> -g -p 8888 -u /dev/USB0
```

This command starts the AJIT processor in debug mode, and uses the serial device /dev/USB0 for the debug connection. Further, it uses port 8888 to connect the AJIT processor debug UART to the remote GDB client.

Note that the processor core has a second UART which can be accessed as the serial device (for prints) by connecting it to a second terminal program. Thus, we have two terminals connected to the processor core, one a debug terminal and one an I/O terminal.

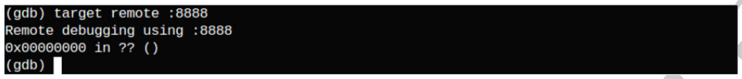
9.2 Starting a remote GDB session

After the processor has been started in DEBUG mode, it will wait for a connection from a remote GDB client. To start the remote GDB client, go to the debug



```
GNU gdb (GDB) 7.6.2
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-unknown-linux-gnu --target=sparc-buildroot-linux-uclibc".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/titto/AJIT_Project/AjitRepoV2/processor/C/debugger/tests/output/Watchpoints.elf...done.
(gdb)
```

Figure 9.1: GDB session: initial prompt



```
(gdb) target remote :8888
Remote debugging using :8888
0x00000000 in ?? ()
(gdb)
```

Figure 9.2: GDB session: program prompt

terminal which is connected to the debug UART and type (at the prompt):

```
$ sparc-linux-gdb <input file>.elf
```

Now the GDB session will start and you will see the messages shown in Figure 9.1

Ensure that the reading of the symbols from elf file succeeds without errors.

Enter the following command at the GDB prompt:

```
(gdb) target remote :8888
```

We use port 8888 to connect the GDB client to the AJIT processor server.

The gdb side will also show the following messages confirming that the connection has been established. Since the connection has now been established, you can control and monitor the program execution on the AJIT processor in real time.

9.3 Basic GDB command support

The following GDB commands are suported: list, continue/next, break-points, watch-points,

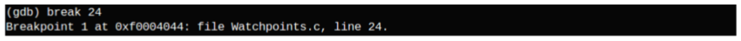
9.3.1 The list command

You can list the code being executed by the processor using the list command

```
(gdb) list
```

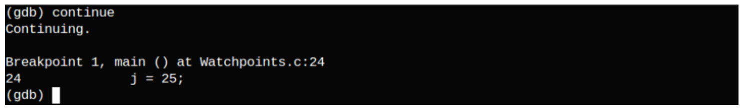
If you wish to see the assembly code, you can use

```
(gdb) disassemble
```



```
(gdb) break 24
Breakpoint 1 at 0xf0004044: file Watchpoints.c, line 24.
```

Figure 9.3: GDB break-points



```
(gdb) continue
Continuing.

Breakpoint 1, main () at Watchpoints.c:24
24              j = 25;
(gdb)
```

Figure 9.4: GDB break-point hit

9.3.2 Continue/Next

When the execution is stopped due to some reason, you can ask the processor to continue (use continue or c) with these commands.

```
(gdb) continue
```

The next (or n) command also asks the processor continue, but stops it after executing one line of code.

```
(gdb) next
(gdb) n
```

9.3.3 Setting and removing breakpoints

Breakpoints are useful for stopping the execution on reaching specific points in the source code, and probe the processor for information. They can be set using the following commands.

```
(gdb) break <function_name>
(gdb) break <line_number>
(gdb) break <filename : function_name>
(gdb) break <filename : line_number>
(gdb) break *<address>
```

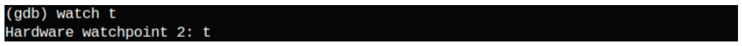
See Figure 9.3 for an illustration.

The AJIT processor will stop execution when a breakpoint is hit, and you will be notified in gdb, as shown in Figure 9.4. Breakpoints can be listed by the following command.

```
(gdb) info breakpoints
```

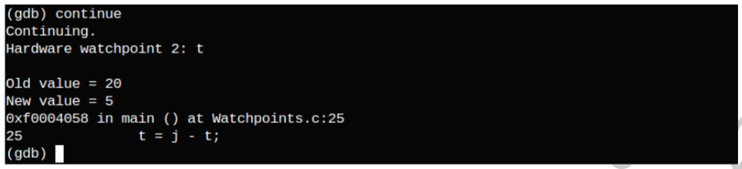
Break points can also be deleted.

```
(gdb) delete break <breakpoint_number>
```



```
(gdb) watch t
Hardware watchpoint 2: t
```

Figure 9.5: GDB watch-point



```
(gdb) continue
Continuing.
Hardware watchpoint 2: t

Old value = 20
New value = 5
0xf0004058 in main () at Watchpoints.c:25
25              t = j - t;
(gdb) 
```

Figure 9.6: GDB watch-point hit

9.3.4 Watchpoints

Watchpoints are useful for stopping the execution on modification of specific variables in the source code. They can be set using the following commands.

```
(gdb) watch [-l|-location] <variable>
(gdb) break [-l|-location] <expression>
```

Ordinarily a watchpoint respects the scope of variables in expression. The `-location` argument tells gdb to instead watch the memory referred to by expression. An example is shown in Figure 9.5. The AJIT processor will stop execution when a watchpoint is hit, and you will be notified in the GDB window (see Figure 9.6).

Watchpoints can be listed by the following command.

```
(gdb) info watchpoints
```

This command can be used to delete specific watchpoints.

```
(gdb) delete watch <watchpoint_number>
```

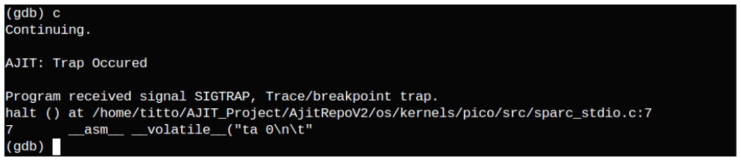
9.3.5 Passing of control to the GDB: break-points, watch-points, interrupts, traps

When GDB asks the processor to continue, it will run until either a break-point/watch-point is hit, or until an exception (trap/interrupt) occurs during execution. From GDB, you can then use the `TBR` register to get information about the trap occurred. One such message is shown in Figure 9.7.

9.3.6 Setting and examining variables/memory/registers

You can modify the values of variables, memory contents and register values with the set command.

```
(gdb) set var <variable> = <value>
(gdb) set $ <register> = <value>
(gdb) set *(int)<memory_address> = <value>
```

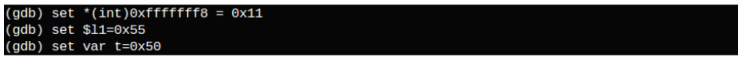


```
(gdb) c
Continuing.

AJIT: Trap Occured

Program received signal SIGTRAP, Trace/breakpoint trap.
halt () at /home/titto/AJIT_Project/AjitRepoV2/os/kernels/pico/src/sparc_stdio.c:7
7       __asm__ __volatile__("ta 0\n\t"
(gdb)
```

Figure 9.7: Trap communicated to GDB



```
(gdb) set *(int)0xfffffff8 = 0x11
(gdb) set $l1=0x55
(gdb) set var t=0x50
```

Figure 9.8: GDB: set variables

This is illustrated in Figure 9.8.

You can display the values of variables, memory contents and register values with the `print` or `p` command.

```
(gdb) print var <variable>
(gdb) print $ <register>
(gdb) print *(int)<memory_address>
```

This is illustrated in Figure 9.9.

9.3.7 Detach

You can stop debugging the program and let the processor continue till it finishes execution, using this command.

```
(gdb) detach
```

This is illustrated in Figure 9.10.

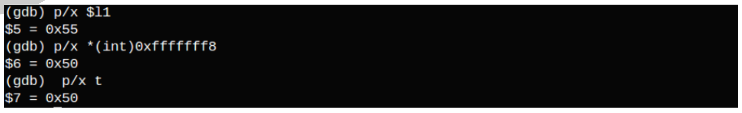
9.3.8 Other useful commands

You can get help on any commands using

```
(gdb) h[elp]
(gdb) h[elp] <command>
```

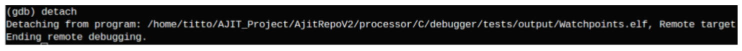
Finish current function, loop, etc. with

```
(gdb) fin[ish]
```



```
(gdb) p/x $l1
$5 = 0x55
(gdb) p/x *(int)0xfffffff8
$6 = 0x50
(gdb)  p/x t
$7 = 0x50
```

Figure 9.9: GDB: print variables



```
(gdb) detach
Detaching from program: /home/titto/AJIT_Project/AjitRepoV2/processor/C/debugger/tests/output/Watchpoints.elf, Remote target
Ending remote debugging.
```

Figure 9.10: GDB: detach

Show lines of code surrounding the current point

`(gdb) l[ist]`

Delete all breakpoints

`(gdb) d[elete]`

Add a list of gdb commands to execute each time a breakpoint is hit

`(gdb) comm[ands] <breakpoint_number>`

9.4 More information

See the GDB manual!

Chapter 10

Processor core add-ons

There are some useful building blocks which can help you put together a system using the generic AJIT processor core. These are collected into a library named GenericCoreAddOnLib, and kept in a file GenericCoreAddOnLib.vhdl.

We describe these add-ons briefly.

10.1 AFB splitter

The AFB splitter is used to divide an AFB bus into two segments based on base address and address mask specification.

The split in the address space is defined by four 36-bit numbers:

```
MIN_ADDR_HIGH_ADDR_SPACE
MAX_ADDR_HIGH_ADDR_SPACE
MIN_ADDR_LOW_ADDR_SPACE
MAX_ADDR_LOW_ADDR_SPACE
```

A 36-bit address X is said to be in the high address space if

$$\text{MIN\_ADDR\_HIGH\_ADDR\_SPACE} \leq X \leq \text{MAX\_ADDR\_HIGH\_ADDR\_SPACE} \tag{10.1}$$

and to the low address space if

$$\text{MIN\_ADDR\_LOW\_ADDR\_SPACE} \leq X \leq \text{MAX\_ADDR\_LOW\_ADDR\_SPACE} \tag{10.2}$$

Note that if there is an address X which either belongs to both spaces or belongs to neither of the two spaces, then the AFB splitter returns an error response.

10.2 AFB SPI Master

The AFB SPI master provides SPI master functionality with an AFB interface. The structure of the AFB SPI master is indicated in Figure 10.2. Using the AFB

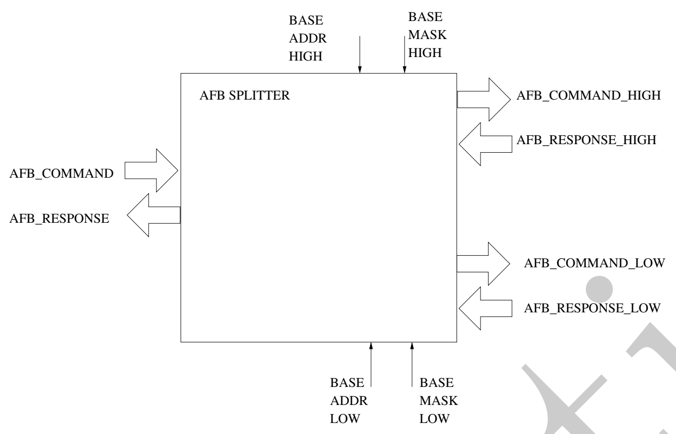


Figure 10.1: AFB Splitter

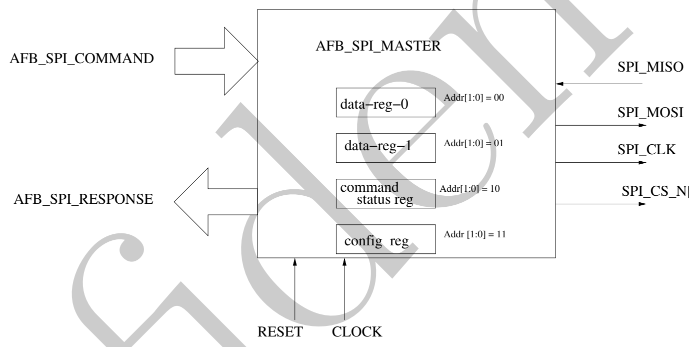


Figure 10.2: AFB SPI Master

bus, it is possible to address the four registers in Figure 10.2 to achieve the SPI master function. The SPI-master implementation is based on an implementation by Hans Huebner (Copyright 2009-2010) from opencores.org.

The registers are addressed 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
[3:0] clock-divide count
```

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).

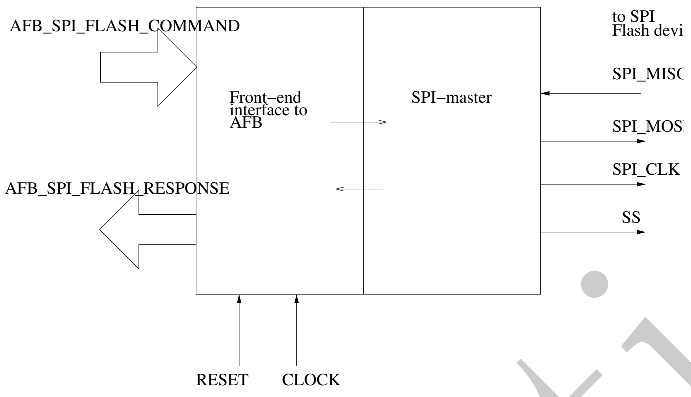


Figure 10.3: AFB SPI Flash controller

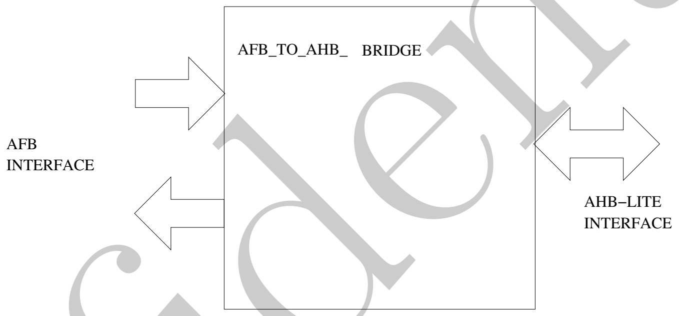


Figure 10.4: AFB to AHB-lite Bridge

10.3 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). Currently, the AFB SPI Flash controller provides read-only access to serial (SPI) flash memory. Alternate resources on an FPGA card can be used to write to the flash memory. The interface of the controller is shown in Figure 10.3. With this controller, the flash memory can be accessed using the normal AFB request response protocol.

10.4 AFB to AHB bridge

The AFB to AHB bridge (Figure 10.4) provides a means of connecting an AHB-lite peripheral to the AFB bus.

10.5 AFB SRAM

A single cycle SRAM (FPGA BRAM compatible) with AFB interface.

10.6 AHB SRAM

A single cycle SRAM (FPGA BRAM compatible) with AHB-lite interface.

10.7 UARTs, TIMERS etc

A simple UART and a count-down interrupting timer device are also available as peripheral devices. These are currently integrated into the generic AJIT processor core, but can also be used as AFB peripherals.

Chapter 11

Tethered PCIE AJIT processor core FPGA evaluation platform

We provide a tethered PCIE based platform for evaluation of the AJIT processor base core. We map the processor base core to an FPGA card with a PCIE interface. Some FPGA SRAM is also integrated with the core. The general arrangement is shown in Figure 11.1. From the host, it is possible to communicate with the debug and serial I/O interfaces on the processor core via the PCIE interface.

The tethered platform uses the RIFFA [3] platform to communicate over the PCIE interface, and has been tested on Xilinx VC709 and KC705 FPGA boards. On the VC709 platform, 4MB of SRAM is available, and on the KC705 platform, 1MB of SRAM is available.

11.1 Driver for tethered PCIE platform

In order to use the tethered PCIE prototype, insert a VC709/KC705 card into a PC (details shown below). The FPGA must be programmed with the appropriate bit file. The processor itself is mapped to FPGA, and the executable accesses the processor via a PCI-e interface.

The driver loads the memory specified by the -m option into the processor memory, and the processor executes the loaded code starting from PC=0x0 until the processor goes into error mode. It is possible to connect a debugger (gdb) to the model and to monitor contents of registers/memory in the processor model.

The model is invoked by

```
ajit_chip_simplified_vhdl_sim_testbench
```

The principal options for the model are

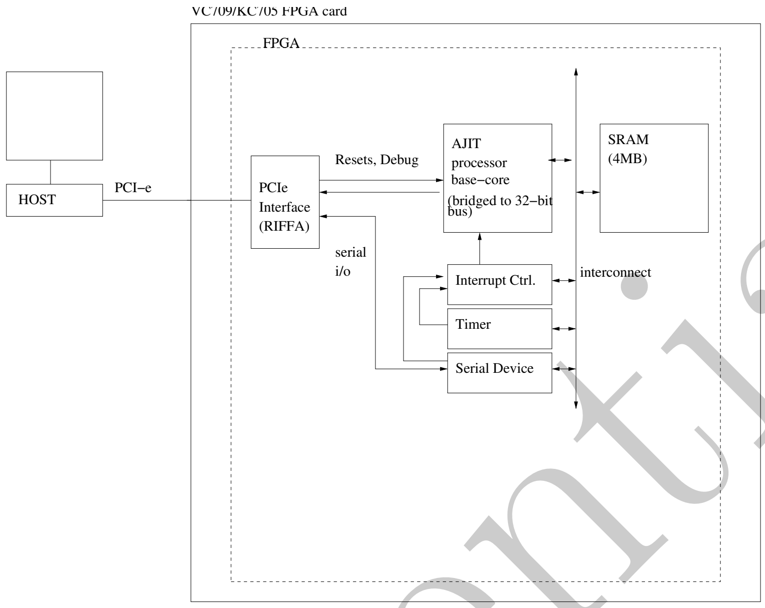


Figure 11.1: The tethered PCIE t AJIT prototype platform

```
-m <mmap-file>
   required, specifies memory-map of processor for this test.
-M
   required, to download the code to the FPGA memory using the
   processor debug interface.
-F
   required, to use the FPGA card.
[-g]
   run the CPU in debug mode.
[-p <gdb-port-number>]
   required with -g, to specify remote debug port.
   Typically, we use port 8888.
[-d]
   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>]
   size of memory is 2**<number-of-address-bits>, default is 32.
[-w <reg-writes-dump>]
```

```
   if specified, a log of all register and memory writes is
   generated.
[-t <trace-file>]
   if specified, produces a detailed activity trace
   for the execution (a big file is generated!).
[-i]
  if specified, spins after processor goes into error mode
  until ctrl-C is pressed.  This is useful for examining
  processor state in debug mode after processor has gone
  to error mode.
[-s]
  run the CPU in single-step mode.
[-v]
  verbose-mode on (trace messages will be logged).
[-I]
  initialize all iunit registers to 0x0.
[-C]
  set cacheable bit in MMU control register.
```

An example:

```
ajit_chip_simplified_vhdl_sim_testbench -m add_test.mmap -M -F -v \
      -d -l add_test.log -r add_test.results
```

This invokes the FPGA model (-F), loads the memory through the debug interface (-M), and sets the verbose flag on so that the loading of the memory is displayed.

Note: With the -F option, you must use the -M option to initialize the memory on the FPGA board.

11.2 Tethered PCIE rocessor evaluation platform details

In order to use the tethered PCIE processor evaluation platform, you need to have the following available.

- PC with Ubuntu 12.04 x86_64, 8GB RAM, Intel(R) Xeon(R) CPU E3-1220 v3 @ 3.10GHz, PCIE slot (full-size).
  - Need to install Xilinx LabTools 14.2 to get drivers for programming FPGA card.
- One of the following:
  - KC705 evaluation card, power adapter and cables from Xilinx (the KC705 kit).

  - VC709 evaluation card, power adapter and cables from Xilinx (the VC709 kit).
  - You may use a different card if you wish. Porting the design to a new card is straightforward.

- Ensure that the evaluation cards are ok.
  - Pass diagnostics.
  - Jumper settings to allow progamming via USB cable.
- AJIT platform distribution bundle (details in Chapter 14).
  - Compilation tools, gdb, bin-utils for Sparc-V8.
  - AJIT evaluation platform simulation tools: C-reference model.
  - FPGA ngc-files/bit-files as relevant.
  - Examples.
  - Documentation.

To setup the platform:

- Perform the Xilinx supplied diagnostic test on card.
- Insert card in to PCI-e slot.
  - Make jumper settings for programming via USB.
  - Connect progamming cable (USB) between card and PC.
  - Connect power supply to card.
- Program card with the bit-file using Xilinx impact.
- Reboot the PC (do not power it off).
- Run ”lspci”, you should be able to see a Xilinx memory controller being detected. If yes, you are ready to go.

In order to run an example, you need to go through the following steps:

- Cross-compilation.
  - sparc-linux-gcc, sparc-linux-as, sparc-linux-ld, sparc-linux-readelf, bin-utils.
  - Custom scripts.
- AJIT platform models: C-reference model, FPGA model.
- Run the application first in C-reference model.
  - Application runs till the processor is put into error mode (the halt condition).

  - Observe post-conditions.
  - Can do remote debugging via sparc-linux-gdb.
- Run the application in the FPGA model.
  - In the FPGA model, the memory map to memory is downloaded to the evaluation board.
  - Application runs until processor is put into error mode.
  - Examine post conditions.
  - Can do remote debugging via sparc-linux-gdb.

These steps are illustrated using examples distributed as part of the platform.

11.3 Compatibility with AJIT debug monitor

The AJIT debug monitor (Chapter 13) supports connections with the tethered PCIE system described in this chapter.

Chapter 12

Building a standalone platform using the generic AJIT core

It is easy to use the generic core together with add on elements to construct a standalone platform. One such platform is shown in Figure 12.1. The AFB splitter, AFB SRAM, and AFB SPI Flash Interface are building blocks that are included in the generic AJIT core distribution (see Chapter 10). The AFB splitter is used to divide the address space into a low space (for Flash, 0x0 to 0x3ffffff) and a high space (for SRAM, 0x40000000 to 0xfffeffff).

This standalone platform can be controlled using

- on board switches for the resets.
- Debug terminal.
- Serial terminal.

The sequence of actions to program and use the standalone platform is

- Generate the FPGA bitfile. The generic AJIT core ngc is provided ready to use, and the addons are provided as VHDL files, in library GenericCoreAddOnLib.
- Program the FPGA.
- Keep the CPU_RESET to '1'.
- Compile and link your program (see Chapter 8) to produce a memory map file and a flash image.
- Program the flash image to the SPI flash.

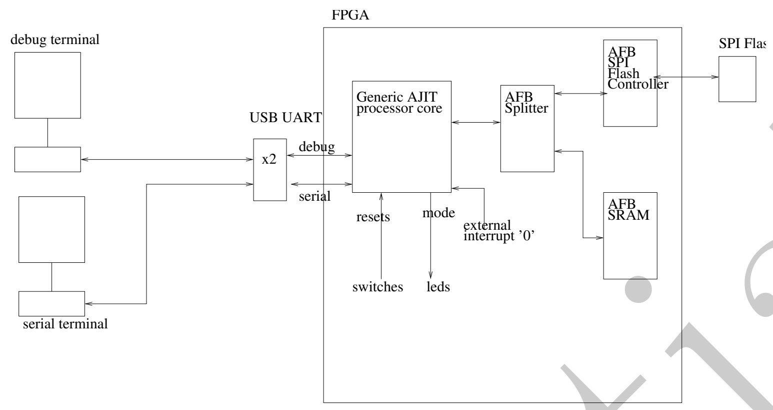


Figure 12.1: A standalone AJIT prototype platform

- Use the ajit_stand_alone_testbench (described in Chapter 5) as the debug connection. You can simultaneously open a serial connection using a second terminal.
- Release the CPU_RESET. Code will be copied from flash to SRAM and then executed from SRAM.

12.1 Driver for AJIT standalone FPGA model with serial UART and debug UART

This driver provides a debug UART connection to the AJIT processor running on an FPGA board. The serial I/O UART needs to be connected to a separate terminal. It is assumed that the FPGA board is programmed using a bit-file before using this model.

As in the other models described in Chapter 5, this loads the memory specified by the -m option into the processor memory, using the debug UART. The processor executes the loaded code starting from PC=0x0 until the processor goes into error mode. It is possible to connect a debugger (gdb) to the model and to monitor contents of registers/memory in the processor model as in the other two cases.

The model is invoked by

```
 ajit_stand_alone_testbench
```

The principal options for the model are

```
[-m <mmap-file>]
   optional, specifies memory-map of processor for this test.
   if omitted, then it is assumed that flash will be used to
   initialize the SRAM.
```

```
-u <serial-dev-for-debug-uart>
   required, a device id (e.g /dev/USB0), which is used to
   download code into the FPGA board SRAM.
[-g]
   run the CPU in debug mode.
[-p <gdb-port-number>]
   required with -g, to specify remote debug port.
[-d]
   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.
```

An example:

```
ajit_stand_alone_testbench -m add_test.mmap -u /dev/USB0
```

This loads the memory map add_test.mmap using the debug UART (/dev/USB0), and sets the verbose flag on so that the loading of the memory is displayed.

12.2 Compatibility with AJIT debug monitor

The AJIT debug monitor (Chapter 13) supports connections with the stand-alone system described in this chapter.

Chapter 13

The AJIT debug monitor utility

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 in Figure 13.1. There are two connections from the user side to the processor model which need to be managed. The first is a debug connection, and the second is a serial I/O connection.

The AJIT debug monitor utility provides a shell which serves as the debug connection.

The Serial I/O connection with the processor model is routed to a console client over TCP/IP using a specified port. A utility such as netcat (nc) can be used to setup a terminal process that uses this TCP/IP port.

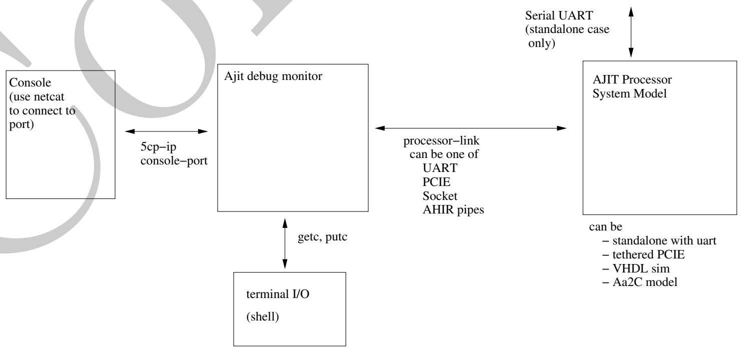


Figure 13.1: The AJIT debug monitor setup

13.1 Synopsys

The AJIT debug monitor utility is invoked at the command line

```
ajit_debug_monitor [-H] [-E] [-A] [-u <tty-dev>] [-v] [-c <console-port>]
```

The options are

-H optional, used when the processor core is modeled by a VHDL simulation.

-E optional, used when the processor core is mapped to an FPGA card with a PCIE interface as the debug connection.

-A optional, used when the processor core is modeled in C.

-u tty-dev optional, when the processor core is mapped to an FPGA board, with a serial device (tty-dev) being used as the debug connection.

-v optional, print more information about what is going on in the background. Also do read-after-write checks while downloading a memory map file (see the "m" command described later).

-c console-port optional, start console to listen on port console-port. This option is relevant only with the -H, -E and -A options.

13.2 AJIT debug monitor commands

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

```
ajit>
```

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 a hex value,
        specified using the 0x format.

        e.g.
            w rst 0x1
        puts the processor in reset mode, and
            w rst 0x0
        brings it out of reset.

rst values constitute a 4-bit field,
with the bit-fields being interpreted as follows:
         [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 processor will run in single-step
                    mode.
         [1]  :  if set, connect to debugger before executing
                    first instruction, after reset is released
         [0]  :  if set, initialize the CPU, Caches, MMU. This
                    must be cleared to start running code.

    r mode
        Read CPU mode.  This is a 4-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.
        e.g.
            w ipc 0x40000000
        means the 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>
        Set the initial NPC at which the processor core starts
        when it comes out of reset.
        e.g.
            w inpc 0x40000004
        means the 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>
        Set the initial PSR at which the processor core starts
        when it comes out of reset.
        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>
        Set 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 Integer 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>
        load 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>
        execute 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 processor core 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.
```

13.3 Setting up the environment

We assume that the processor model is set up and ready to use. This means that one of the following situations is present:

- The processor+system is mapped to an FPGA card, with the debug interface being mapped to PCIE (card must be in PCIE slot, FPGA must be programmed).
- 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.
- The processor+system is running in a VHDL simulation. The debug connection is via a TCP/IP socket.
- The processor+system are modeled by a C model (more precisely, an AA2C model). The debug connection is via the AHIR pipe-handler, which is linked to the processor model and the monitor.

13.4 Typical use cycle

In typical use, one follows the sequence (script file shown below):

```
! put the processor in reset
w rst 0x1
! download the memory map (m <mmap-file>)
m ADD.mmap
! set initial values for pc/npc/psr.
w ipc 0x0
w inpc 0x4
w ipsr 0x10c0
! release the reset (w rst 0x0)
w rst 0x0
! keep checking mode (r mode)
r mode
r mode ...
! quit / go back to reset mode and try again.
q
```

You can start a GDB server through the debug monitor. Unfortunately, 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.

```
! put the processor in reset+debug-mode
w rst 0x3
```

```
! download the memory map (m <mmap-file>)
m ADD.mmap
! set initial values for pc/npc/psr.
w ipc 0x0
w inpc 0x4
w ipsr 0x10c0
! bring processor out of reset, but in debug-mode
w rst 0x2
! switch to gdb..
! start GDB server on port 8888
g start 8888
! From here on, the debug monitor should
! be quiet until the  debugging session is
! over.
! when you are done, type
g stop
! MUST quit.
q
```

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

Chapter 14

AJIT processor platform distribution

The platform is distributed using the following directory structure:

```
top-directory
    see README

    docs/
      this document and other useful documents

    tools/
         scripts/
            lots of useful scripts.  Most important
            is compileToSparc.py
         ajit_access_routines/
            routines which allow the programmer
            to access devices, mmu etc.
         genVmapAsm/
            utility for generating virtual-physical
            mapping assembly code for bare-metal apps.
         flash_image/
            utility for generating flash image.
         minimal_printf_timer/
            basic printf for bare metal using serial device.
         cross-compiler/
            cross-compiler, debugger, linker etc.
         glibc (or uclibc)/
            gnu utils, including math etc.

     ngc/
          NGC file for generic core
```

```
vhdl/
     Important VHDL libraries.

bitfiles/
     Tethered PCIE platform bitfiles.

ahir_release/
     AHIR tools from IITB.
```

This chapter is under construction.

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.