# Using the AJIT processor: a short course

Madhav Desai
Department of Electrical Engineering
IIT Bombay

May 6, 2024

Some indispensable documentation:

- The SPARC V8 instruction set architecture (ISA) document.
- The SPARC V8 assembler manual.
- The SPARC application binary interface document.
- A description of the SPARC stack and its manipulation.
- A description of the AJIT processor and its support infrastructure.
- All these documents are available in the references folder.

SPARC–V8
ISA

AJIT
SPARC–V8
IMPL.

AJIT
peripherals

SPARC–V8
Assembly
Ref.

AJIT
tool–chain

AJIT SOC
build elements

SPARC–V8
Appl.
Binary
Interface (ABI)

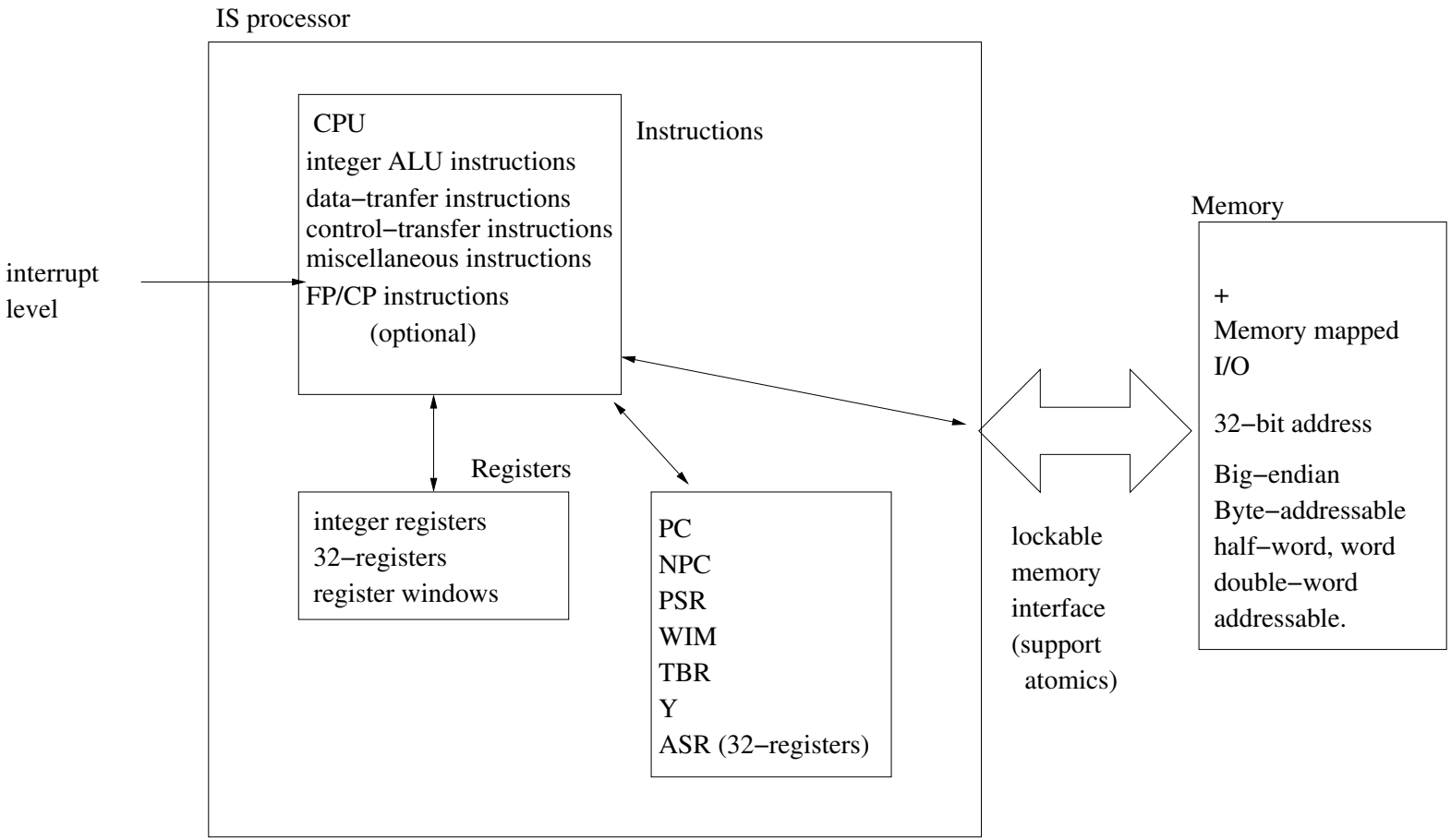
CORTOS2

AJIT
FPGA
prototype

# Highlights

- 32-bit RISC load-store ISA.
- 8/16/32/64 bit integer data types.
  - byte, signed-byte, half-word, signed-half-word, word, signed-word, double-word.
- 32 programmer visible registers
  - Organized into windows.
- ALU, control-transfer, data-transfer, miscellaneous, floating-point (optional), coprocessor (optional) instruction classes.
- 8/16/32/64 bit load/store operations.
- Interrupts, hardware-exceptions, software exceptions.
- SPARC reference MMU specification (optional).
- Delayed control-transfer instructions.
- Interrupts, hardware traps, software traps.

# SPARC-V8 instruction set architecture: overview



# SPARC-V8 instruction set machine behaviour

```
PC := 0, NPC := 4  (others := 0)
while (1) {
  Instr := fetchInstruction (PC)
  NNPC  := executeInstruction (PC, Instr)
  PC    := NPC
  NPC   := NNPC
}
```

# SPARC-V8 instruction set machine behaviour: delay slot

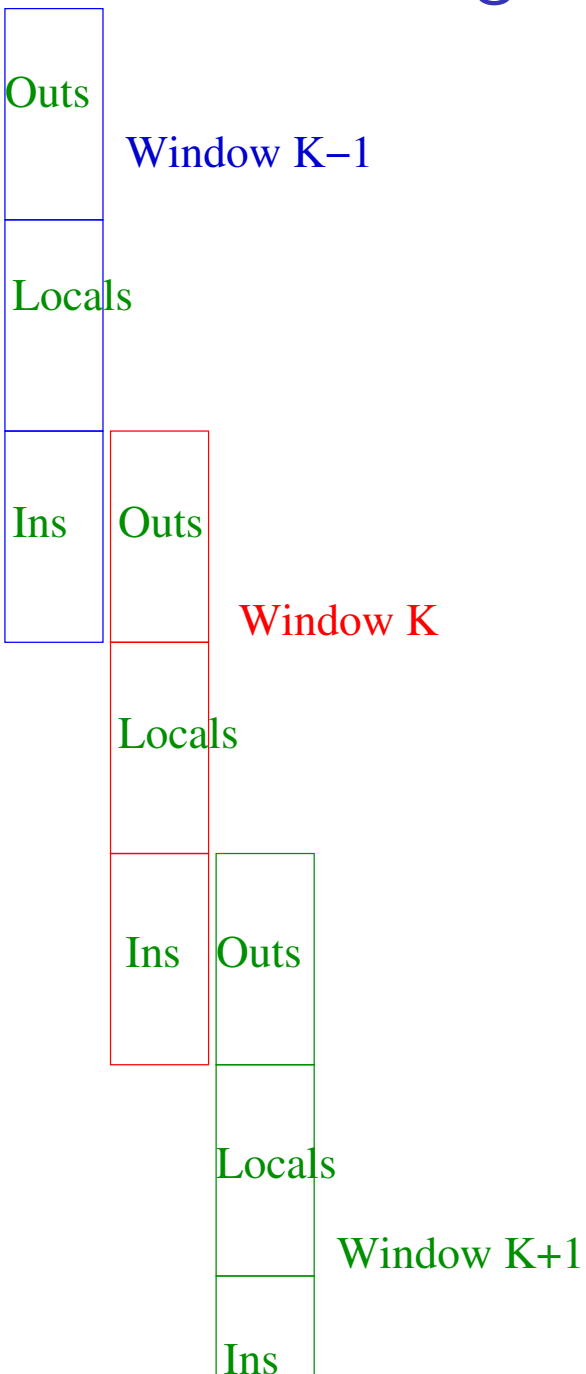
```
------------------------------
Given PC, NPC we execute
------------------------------
     Instruction at PC
          (gives NNPC)

     Instruction at NPC
          (irrespective of
            NNPC)
------------------------------
For the instruction at PC,
the instruction at NPC is
called the delay-slot
instruction.
------------------------------
```

# SPARC-V8 instruction set architecture: registers

- 32 programmer visible 32-bit integer unit registers.
  - Eight global registers $g0, g1, \ldots, g7$, also named $r0, r1, \ldots, r7$.
    - Register g0 (also known as r0) is always 0.
  - Eight out registers $o0, o1, \ldots o7$ (also named $r8, r9, \ldots, r15$).
  - Eight local registers $l0, l1, \ldots l7$ (also named $r16, r17, \ldots, r23$).
  - Eight in registers $i0, i1, \ldots i7$ (also named $r24, r25, \ldots, r31$).
- Register windows: a cache of registers.
  - From 2 to 256 windows:
  - Each window consists of 8-out, 8-local and 8-in registers.
  - Window $k$ overlaps with window $k-1$, with outs of window $k$ being the same as the ins of window $k-1$.
  - The AJIT processor implements 8 windows.
- 32 programmer visible floating point registers (if FP-unit is present).
- System registers: processor state register (PSR), window invalid mask register (WIM), trap base register (TBR), Y register, 32 ASR registers, floating point status register (FSR).

# Visualization of register windows



# Integer unit registers: register windows

- At any point in time, a particular register window is active. Call this the current register window.
  - The index of the current register window is kept in PSR[4:0].
- The programmer sees the 8 global registers, and the 24 window registers (8 outs, 8 locals, 8 ins).
- The special SAVE/RESTORE instructions are used to change the current window. The SAVE instruction decrements the index of the active window so that what used to be "out" registers are now seen as "in" registers. The RESTORE instruction increments the index active window so that the previous "in" registers now become the "out" registers.
- Windows thus provide a low overhead mechanism to transfer arguments during function calls.
- What if we exhaust all windows?

# Processor status register

| Impl | ver | icc | reserved | EC | EF | PIL | S | PS | ET | CWP |
|---|---|---|---|---|---|---|---|---|---|---|
| 31:28 | 27:24 | 23:20 | 19:14 | 13 | 12 | 11:8 | 7 | 6 | 5 | 4:0 |

# Condition codes

- In the integer unit:
  - The PSR register implements four condition code bits in field PSR[23:20].
  - The condition code bits are N (negagive), Z (zero), V (overflow), and C (carry) .
  - Condition codes are set by ALU instructions (e.g. ADDcc, SUBcc, MULcc etc.).
- In the floating point unit (when present):
  - The FSR register (FSR[11:10]) implements a 2-bit condition code which remembers the status of the last floating point compare.
- Condition codes are used for branching.

# User and supervisor modes

The processor can be in one of two modes

- In user mode, bit S is 0. A subset of instructions can be executed in user mode. When in user mode, an attempt to execute an instruction outside this subset results in a trap.
- In supervisor mode, bit S is 1. All instructions are executable in supervisor mode.

# Window-invalid-mask register

| Zero | Window-invalid-bits |
|---|---|
| 31:8 | 7:0 (for 8-window machine) |

# Trap-base register

| Trap-base-addr (TBA) | Trap-type (TT) | Zero |
|---|---|---|
| 31:12 | 11:4 | 3:0 |

Address of trap-handler = [TBA TT Zero4]

256 vectored traps:

128 HW (including 15 Interrupts)

128 SW.

# Ancillary state registers (ASR's)

- 32 32-bit registers.
- Use of these registers is implementation dependent.
  - ASR31 and ASR30 (concatenated) provide a 64-bit cycle counter (in the AJIT processor).
  - ASR29 provides core-id, cpu-id (in the AJIT processor).
  - ASR28 provides a thread descriptor which encodes hardware parameters such as cache size, associativity etc (in the AJIT processor).

# Y register

- Used as an intermediate register in step-by-step multiplication.
- Forms the upper half of the dividend in a divide operation.

# SPARC-V8 instruction set architecture: memory organization

- 32-bit virtual addresses are generated by the SPARC CPU.
- In addition to address, a memory access is specified by an 8-bit address space identifier (ASI). The common ASI identifiers are:
  - ASI=0x8 means that the access is an instruction access with user mode privileges.
  - ASI=0x9 means that the access is an instruction access with supervisor mode privileges.
  - ASI=0xA means that the access is a data access with user mode privileges.
  - ASI=0xB means that the access is a data access with supervisor mode privileges.
- Other ASI identifiers specify access to specific locations in the processor system, such as for example, registers in the memory management unit etc.
- ASI identifiers from 0x20 to 0x2F are called bypass ASI identifiers, and provide a mechanism of direct addressing of physical memory (without virtual to physical translation).

# SPARC-V8 instruction set architecture: big-endian-ness

- Units of data are bytes (7:0), half-words (15:0), words (31:0) and double-words (63:0).
- Addresses to bytes are arbitrary integers, addresses to half-words are multiples of 2, addresses to words are multiples of 4, and addresses to double words are multiples of 8.
- For a double word address (e.g. 128), the most significant byte in the double word has address 128, the next most significant has address 129, and so on. Thus, the data is kept in big-endian format.
- Unaligned accesses are treated as exceptions and need to be handled by an appropriate trap handler.

# SPARC-V8 instruction set architecture: instruction classes

- Integer ALU.
- Control transfer.
- Data transfer: byte/half-word/word/double-word load/store.
- Miscellaneous: swap, read/write status registers, save/restore windows, etc.
- Floating-point instructions (optional).
- Coprocessor instructions (optional).
- Vectored trap/interrupt handling.
- Reference memory management unit specification (optional).

# Integer ALU instructions

- ADD/SUB/UMUL/SMUL/UDIV/SDIV etc., each with set CC options.
- AND/OR/ORN/NAND/NOR/XOR/XNOR etc., each with set CC options.
- Shifts: SLL/SRL/SRA
- Trap on overflow instructions.

# Integer ALU instructions: encoding in assembly

```
add %g1, %g2, %g3     ! <g3> = <g1> + <g2>
add 0x5, %g2, %g3     ! <g3> = 0x5 + <g2>

addcc %g1, %g2, %g3   ! <g3> = <g1> + <g2>
                      ! set CC in PSR.

etc..
```

# Control transfer instructions

- In the integer unit:
  - Bicc instructions (16 of them, depending on the condition codes being checked), CALL, JMPL, RETT intructions.
- In the floating point unit:
  - FBfcc instructions (16 of them, depending on the FP condition codes).
- Will be described in detail later.

# Control transfer instructions: branch with delay slot

```
  bne branch_label      ! branch to branch_label:
  mov 0x0, %g1          ! this instruction is
                        ! executed next

..

branch_label:
 mov 0x1, %g2           ! executed next to next.
  ....
```

# Control transfer instructions: conditional branch with delay slot and annul

```
  bne,a branch_label   ! branch to branch_label:
  mov 0x0, %g1         ! this instruction is
                       ! executed if branch is
                       ! taken, else NOP.

..

branch_label:
 mov 0x1, %g2          ! executed next to next.
```

# Control transfer instructions: unconditional branch with delay slot and annul

```
  ba,a branch_label   ! branch to branch_label:
  mov 0x0, %g1        ! executed if branch is
                      ! not taken, else NOP.

..

branch_label:
 mov 0x1, %g2          ! executed next to next.
```

# Control transfer instructions: CALL, JMPL (jump and link), RETT (return from trap)

```
call subroutine_label
mov 0x1, %g2                ! delay slot instr.
                            ! executed next (delay slot).

jmpl %r18+%r0, %r10     ! jump to PC = <r18> + <r0>
                        ! and store PC in <r10>
mov 0x, %g2             ! executed next (delay slot)

rett ...  will discuss later.
```

# Data transfer (between registers and memory) instructions

- Load/Store between registers and memory.
- Integer registers and memory: byte/half-word/word/double-word loads and stores.
- Floating point registers and memory: word/double-word loads and stores.
- Load/Store alternate: Up to 256 address spaces can be specified.

# Load/Store examples

```
ld [%g1 + %g3], %g2   ! load word from
                      ! memory location (<g1> + <g3>)
                      ! into g2.
ldub [%g1], %g2 ! load unsigned byte
                ! from mem[<g1>]
                ! into g2 (no sign-extend)

st %g2, [%g1 + 0x8]
                ! store word from
                ! g2 into mem[<g1> + 0x8]
etc..
ld, ldd, ldub, ldsb
st, std, stub, stsb
```

# Atomic instructions: SWAP, LDSTUB

```
swap [%g3], %g2   ! tmp = <g2>
                  ! <g2> = mem[<g3>]
                  ! mem[<g3>] = tmp
                  ! ATOMIC

ldstub [%g3], %g2 ! same as above
                  ! but with bytes
                  ! instead of words
                  ! ATOMIC
```

# Save/Restore instruction for changing the current window

```
mov 0x2 %o0              ! we are in window k
save                     ! change window to k-1

add %i0, %i0, %i1        ! %i0 in window k-1
                         ! is same as %o0 in k

                         ! i1 holds value 4.

restore                  ! back to window k
                         ! o0 holds 0x2,
                         ! o1 holds 0x4

add %o0, %o1, %o2        ! o2 evaluates to 0x6
```

# FP instructions (optional)

- 32 32-bit registers (f0, f1, ..., f31).
- Can be viewed as 16 64 registers [f0 f1], [f2 f3] etc.
- Load/store between FP registers and memory.
- Single and double precision add, subtract, multiply, divide, square-root, compares.
- IEEE 754 exception model.
- large number of open op-codes for adding new instructions.

# Coprocessor instructions (optional)

- 32 32-bit registers (f0, f1, ..., f31).
- Can be viewed as 16 64 registers [f0 f1], [f2 f3] etc.
- Load/store between CP registers and memory.
- Coprocessor branch instructions.
- Lots of open op-codes to specify special instructions.

# Exceptions specified in the SPARC-V8 ISA

- Synchronous exceptions: those that take place at known instructions in the program being executed.
  - Hardware exceptions (e.g. divide by zero).
  - Software exceptions caused by explicit trap instructions.
- Asynchronous exceptions: those that take place without any coordination with the program being executed.
  - Interrupts!

# Synchronous hardware exceptions

| Hardware-trap | Id |
|---|---|
| Instruction-access-exception (IAE) | 0x1 |
| Illegal-instruction (II) | 0x2 |
| Privileged-instruction-trap (PI) | 0x3 |
| Floating-point-disabled (FPD) | 0x4 |
| Window-overflow-trap (WOF) | 0x5 |
| Window-underflow-trap (WUF) | 0x6 |
| Unaligned-address-trap (UA) | 0x7 |
| Floating-point-trap (FPE) | 0x8 |
| Data-access-exception (DAE) | 0x9 |
| Tag-overflow-trap (TOF) | 0xA |
| Coprocessor-disabled-trap (CPD) | 0x24 |
| Integer-divide-by-zero (IDZ) | 0x2A |

# Synchronous hardware exceptions

- These occur due to exceptional conditions arising during the execution of the program.
- When such an exception occurs, the processor will jump to a trap-handler in a vectored manner (will be described below). The address of the trap handler is computed using the trap-base register and the ID of the trap.

# Synchronous software exceptions

- The SPARC-V8 ISA includes 16 "trap on integer condition code" instructions.
- For example, the trap-always instruction is specified as

  `ta <trap-type>`

  where trap-type specifies a number between 0 and 63 (inclusive). Whenever the ta instruction is executed, it triggers an exception with Id trap-type[7:0] + 0x80. The trap handler address is then calculated using this Id.
- There are 16-possible trap on integer condition code (Ticc) instructions..

```
ta      trap-always
tn      trap-never
tne     trap-not-equal
te      trap-if-equal
tg      trap-if-greater
tle     trap-less-equal
...etc..
```

  Each specifies the trap-type field, which is used to calculate

# Asynchronous exceptions: interrupts

- 15 interrupt levels are specified.
- An interrupt with level $N$, $1 \leq N \leq 15$ is mapped to exception Id $0x10 + N$.
- The Id corresponding to the interrupt is used to calculate the address of the trap handler.
- Interrupt 15 is a non-maskable interrupt.
- At any point, only those interrupts whose level is greater than the PSR PIL field (PSR[11:8]) will have effect. This is a simple way to mask off interrupts (except the non-maskable interrupt).

# Getting to the trap-handler

- Suppose that an exception with Id=$N$ has occurred. This $N$ will be an 8-bit number.
- The address of the exception handler (call it $A$) is

  `Concatenation of Trap-base-register [31:12], $N$, 0x0`
- The processor executes a SAVE instruction.
- The processor saves the current PC in the l1 register, and the current NPC in the l2 register (in the new window, post SAVE).
- The PSR S bit is saved into the PS bit, and the S bit is set to 1.
- The PSR enable-traps bit is set to 0.
- The processor jumps to the exception handler address, that is, it considers $PC = A$ and $NPC = (A + 4)$.
- The return from the trap handler occurs using a combination of the JMPL, RETT instructions (will be clearer when we show some examples).

# A look at the trap handler

- Vectored trap handlers.
- Specific trap handler: window overflow trap handler.
  - Use of JMPL/RETT to return from trap.

# A look at the generic interrupt/trap handlers

- Save the state of the currently running program.
- Run the interrupt/trap handler.
- Recover the state of the currently running program.
- JMPL/RETT and continue the currently running program.

# Traps on trap not allowed unless traps are enabled.

- If the processor sees a synchronous exception when the enable-traps bit is set to 0 in the PSR, then this is considered an irrecoverable error and the processor goes into error mode, and halts.
  - When this happens, the only way to restart the processor is by applying a power-on reset.
- When traps are disabled, interrupts are ignored.

# What is an ABI

- Standard specification which describes
  - Roles played by specific registers: for example stack-pointer, frame-pointer etc.
  - Conventions for passing arguments between caller and callee routines.
  - Organization of the stack.
- Important to understand how the compiler uses the registers, especially when mixing assembly and C code.
- This specification is described in the SPARC ABI document.

# Use of registers (taken from sparc_abi.pdf document)



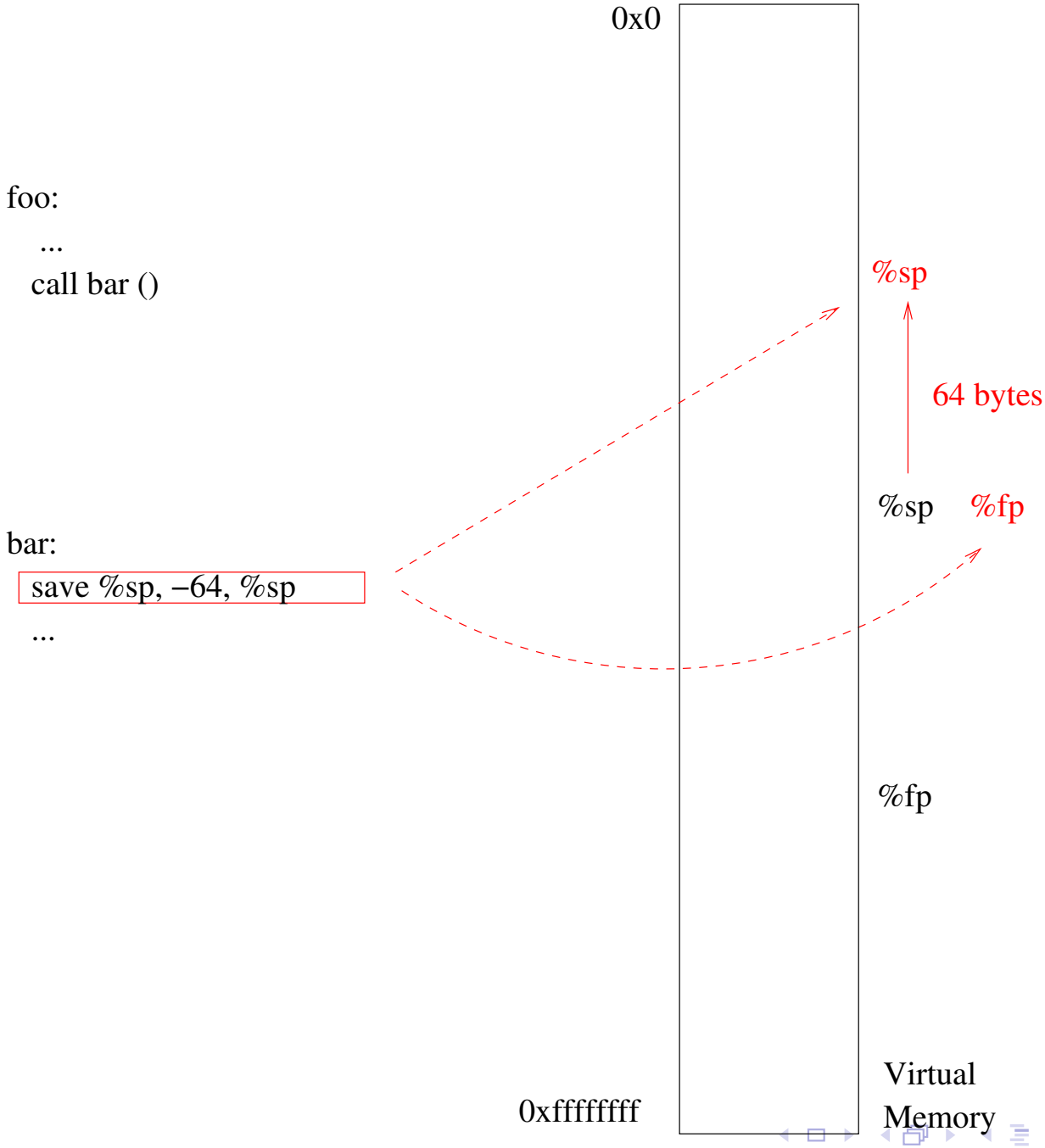
**Figure 3-14: A Function's Window Registers**

| Type | Name | | Usage |
|---|---|---|---|
| *in* | | %i7 %r31 | return address – 8 † |
| | %fp, | %i6 %r30 | frame pointer † |
| | | %i5 %r29 | incoming param 5 † |
| | | %i4 %r28 | incoming param 4 † |
| | | %i3 %r27 | incoming param 3 † |
| | | %i2 %r26 | incoming param 2 † |
| | | %i1 %r25 | incoming param 1 † |
| | | %i0 %r24 | incoming param 0, † outgoing return value |
| *local* | | %l7 %r23 | local 7 † |
| | | %l6 %r22 | local 6 † |
| | | %l5 %r21 | local 5 † |
| | | %l4 %r20 | local 4 † |
| | | %l3 %r19 | local 3 † |
| | | %l2 %r18 | local 2 † |
| | | %l1 %r17 | local 1 † |
| | | %l0 %r16 | local 0 † |
| *out* | | %o7 %r15 | address of `call` instruction, ‡ temporary value |
| | %sp, | %o6 %r14 | stack pointer † |
| | | %o5 %r13 | outgoing param 5 ‡ |
| | | %o4 %r12 | outgoing param 4 ‡ |
| | | %o3 %r11 | outgoing param 3 ‡ |
| | | %o2 %r10 | outgoing param 2 ‡ |
| | | %o1 %r9 | outgoing param 1 ‡ |
| | | %o0 %r8 | outgoing param 0, ‡ incoming return value |

**Figure 3-15: A Function's Global Registers**

| Type | Name | | Usage |
|---|---|---|---|
| | %g7 | %r7 | global 7 (reserved for system) |
| | %g6 | %r6 | global 6 (reserved for system) |
| | %g5 | %r5 | global 5 (reserved for system) |
| | %g4 | %r4 | global 4 (reserved for application) |
| | %g3 | %r3 | global 3 (reserved for application) |
| | %g2 | %r2 | global 2 (reserved for application) |
| | %g1 | %r1 | global 1 ‡ |

# The stack


0x0
foo:
...
call bar ()
%sp
64 bytes
%sp
%fp
bar:
save %sp, –64, %sp
...
%fp
0xffffffff
Virtual
Memory

# Stack frame

```
%fp + 92        in-args 6 and higher
%fp + 68        in-args 0 to 5 (caller writes)
%fp + 64        struct/union return-pointer
%fp + 0         window save area.
... unspecified number of words....
%sp + 92        out-args 6 and higher
%sp + 68        out-args 0 to 5 (callee writes)
%sp + 64        struct/union return-pointer
%sp + 0         window save area.
```

# Special registers

| | |
|---|---|
| %sp (%o6) | stack-pointer |
| %fp (%i6) | frame-pointer |
| %i0 %o0 | integer arg 0, pointer to return struct |
| %i7 and %o7 | return address for call |
| %i1-%i5 | integer args 1-5 |
| %l0 to %l7 | free-to-use |
| %l1, %l2 | store pc, npc on trap. |
| %g0, %g1 | free-to-use |
| %g2,%g3,%g4 | application use |
| %g5,%g6,%g7 | reserved for system software |
| %o1-%o5 | integer args 1-5 |

# Function prologue and epilogue

```
foo:
    save %sp, -80, %sp
    ...
    ...
    jmpl %i7+8, %g0
    restore %g0, 0, %g0
```

Note: the save and restore may be omitted for leaf routines.

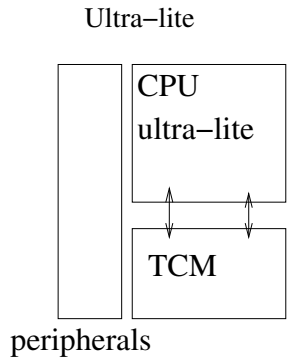
# More in SPARC ABI document (must read!)

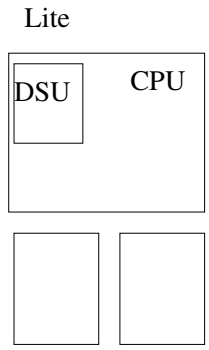
- data structure packing.
- details about passing arguments (more cases).
- operating system interface (System V).
- standard trap handling (signals).
- process initialization.
- dynamic stack allocation.
- file formats for executables (ELF).
- etc.

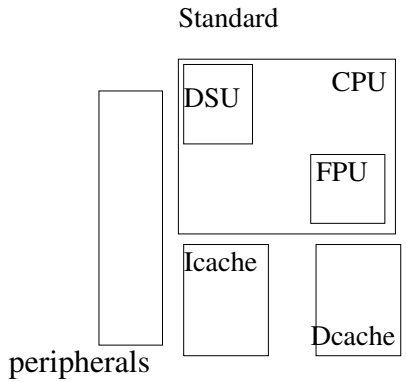
# AJIT core family

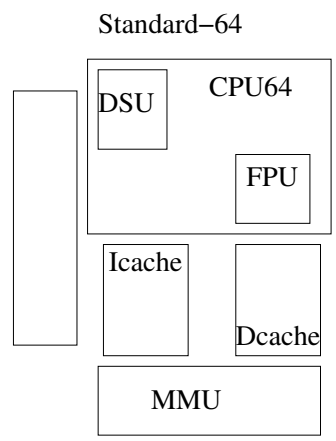
- The AJIT family of processors consists of various implementations of the SPARC-V8 instruction set architecture.
- The simplest implementation is a single-core-single-thread processor and the most complex (thus far) is a quad-core-eight-thread implementation.
- For the purposes of this course, we will use a single-core single-thread implementation
- A full tool-chain is available, and hosted on github.
- A cooperative real-time operating system (CORTOS2) has been developed at IIT Bombay and will be used to develop applications.
- Linux is also ported to cores with memory management units.

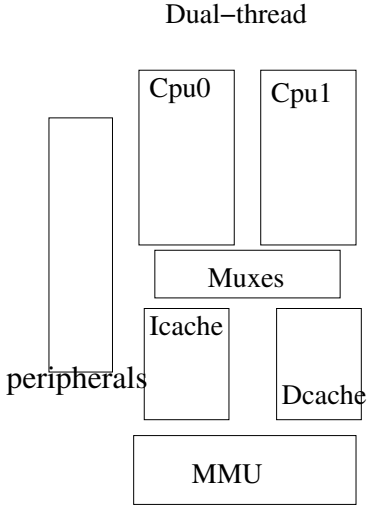
# AJIT processor implementations

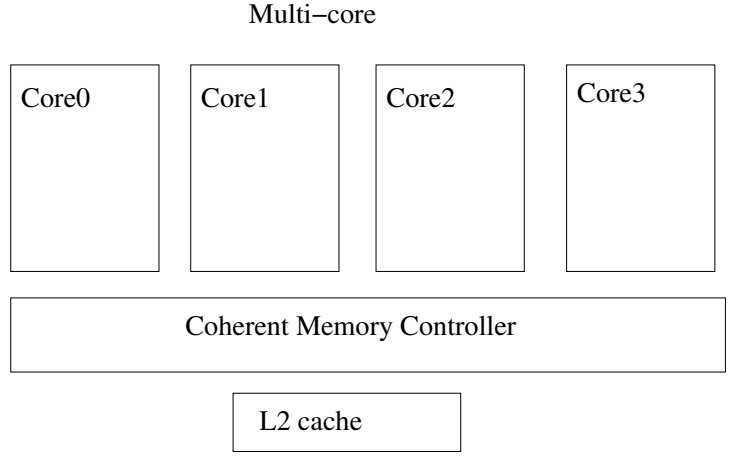


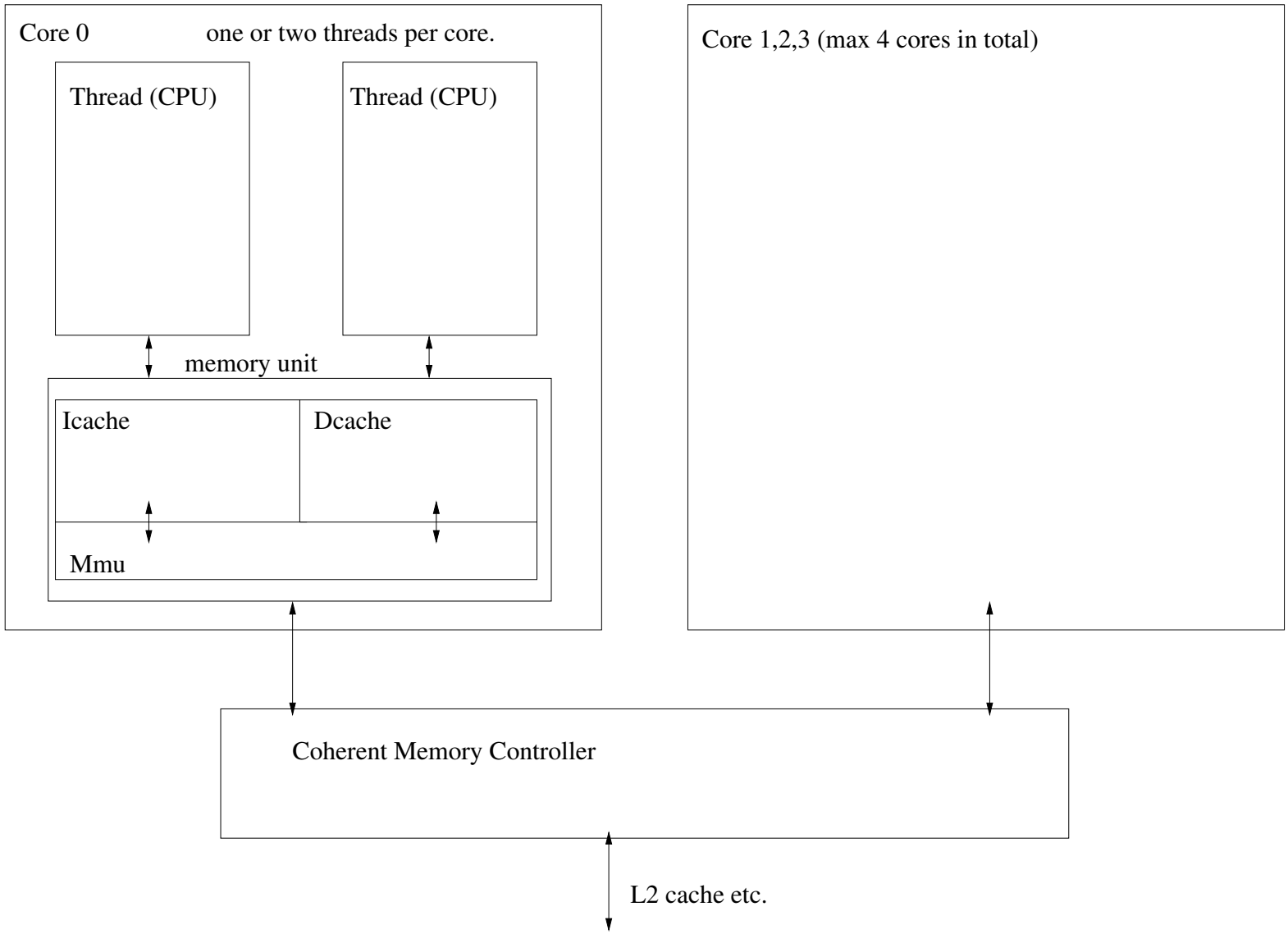



# AJIT processor structure



# AJIT processor structure: salient points

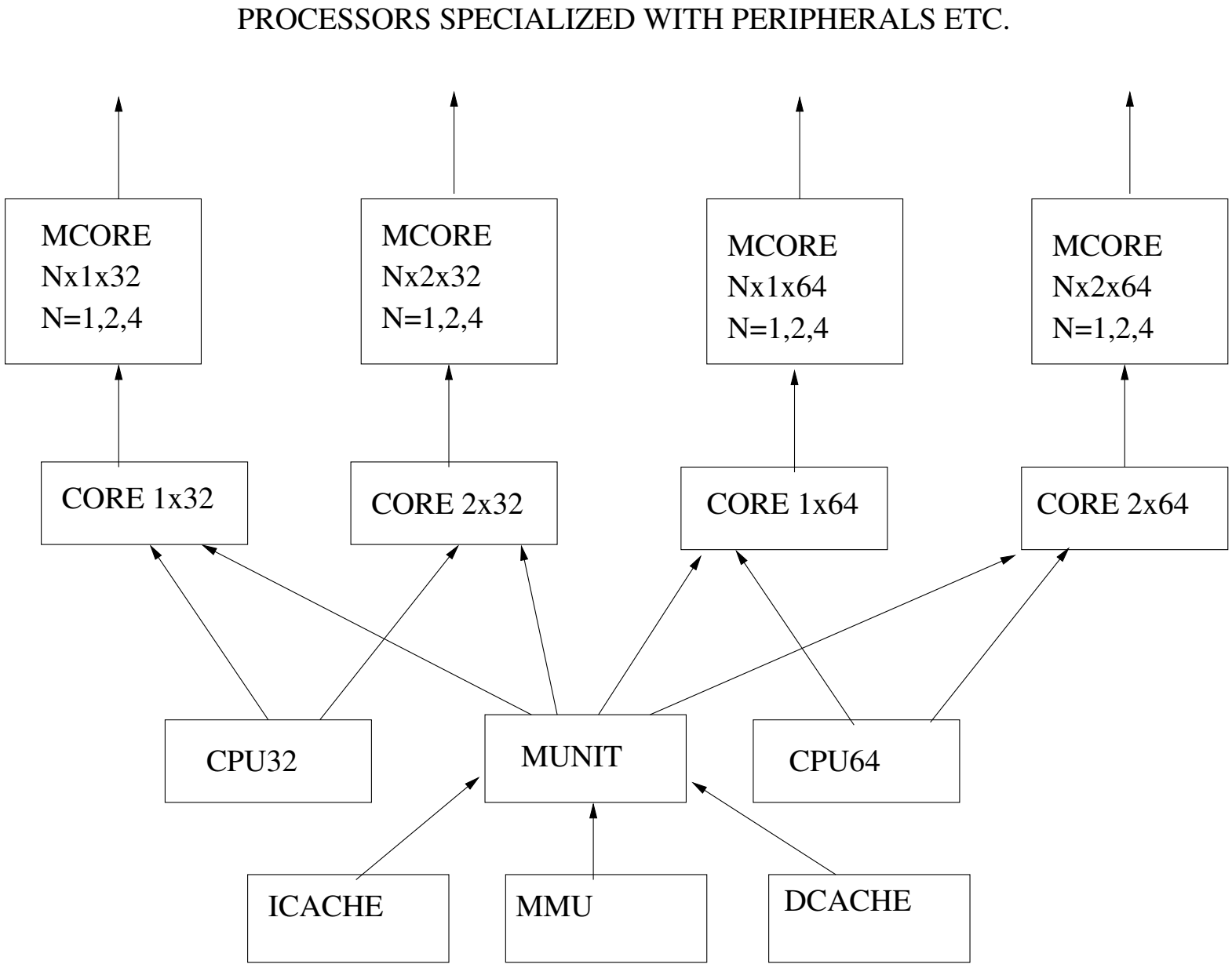
- All the CPU's in AJIT processor cores implement 8 register windows.
- Each processor core has one or two CPU's, and a memory unit.
- The CPU can be configured with or without a floating point unit.
- The memory unit has VIVT instruction and data caches. These can be configured for sizes up to 32 KB, with set associativity up to 8-way.
- The memory unit can be configured with or without a memory management unit (MMU). The MMU, if present, conforms to the SPARC V8 reference MMU standard (described the SPARC V8 ISA document).

# AJIT processor table

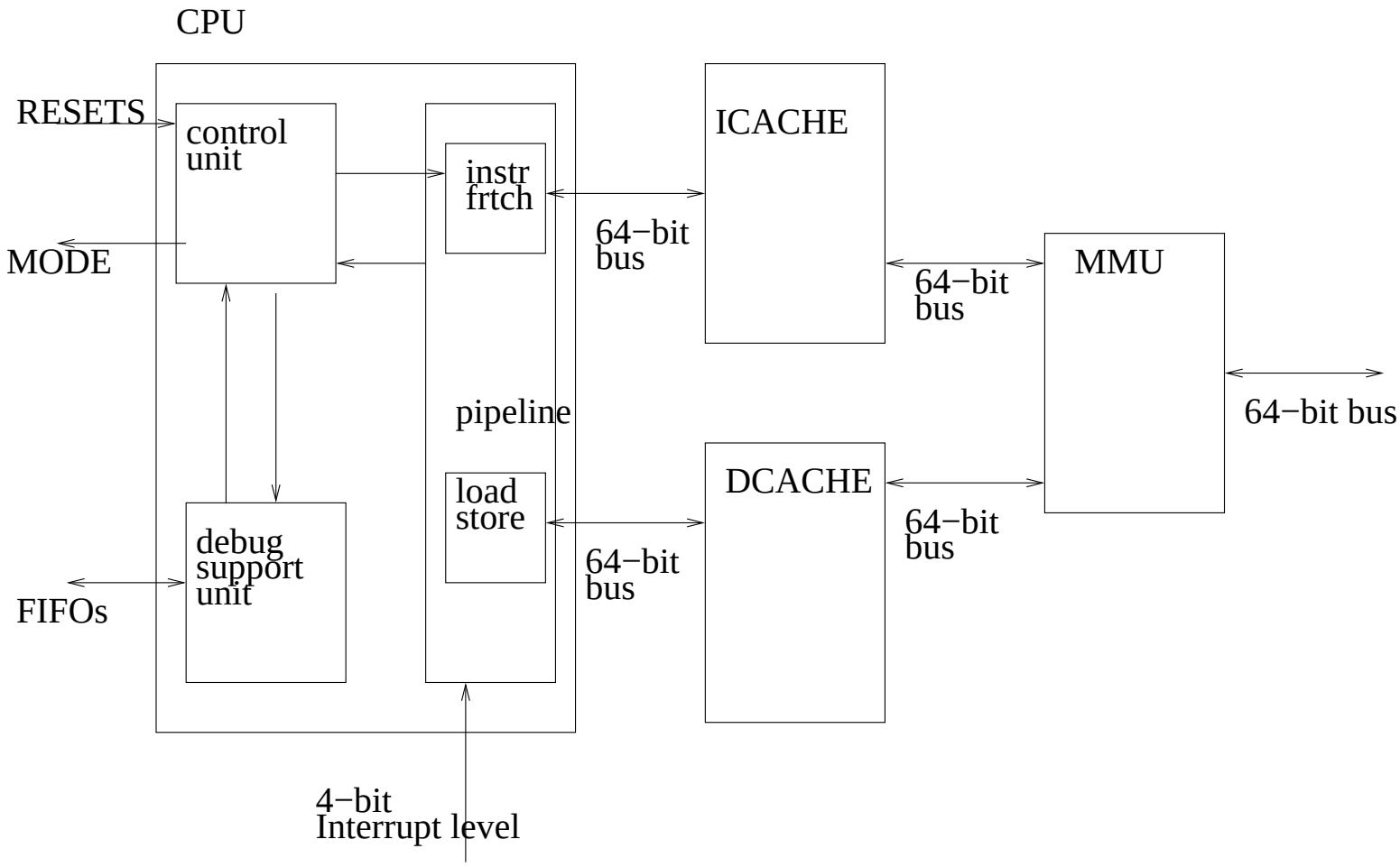
| Name | FPU | ICACHE | DCACHE | MMU | TCM | #T-#C |
|---|---|---|---|---|---|---|
| Ultra-lite | No | No | No | No | Yes | 1-1 |
| Lite | No | Yes | Yes | No | No | 1-1 |
| Standard | Yes | Yes | Yes | Yes | No | 1-1 |
| Standard-64* | Yes | Yes | Yes | Yes | No | 1-1 |
| Multicore | Yes | Yes | Yes | Yes | No | 1/2-1/2/4 |

Note: * the 64-bit extension is a custom AJIT enhancement which provides 64-bit integer operations as well as SIMD integer and float operations.

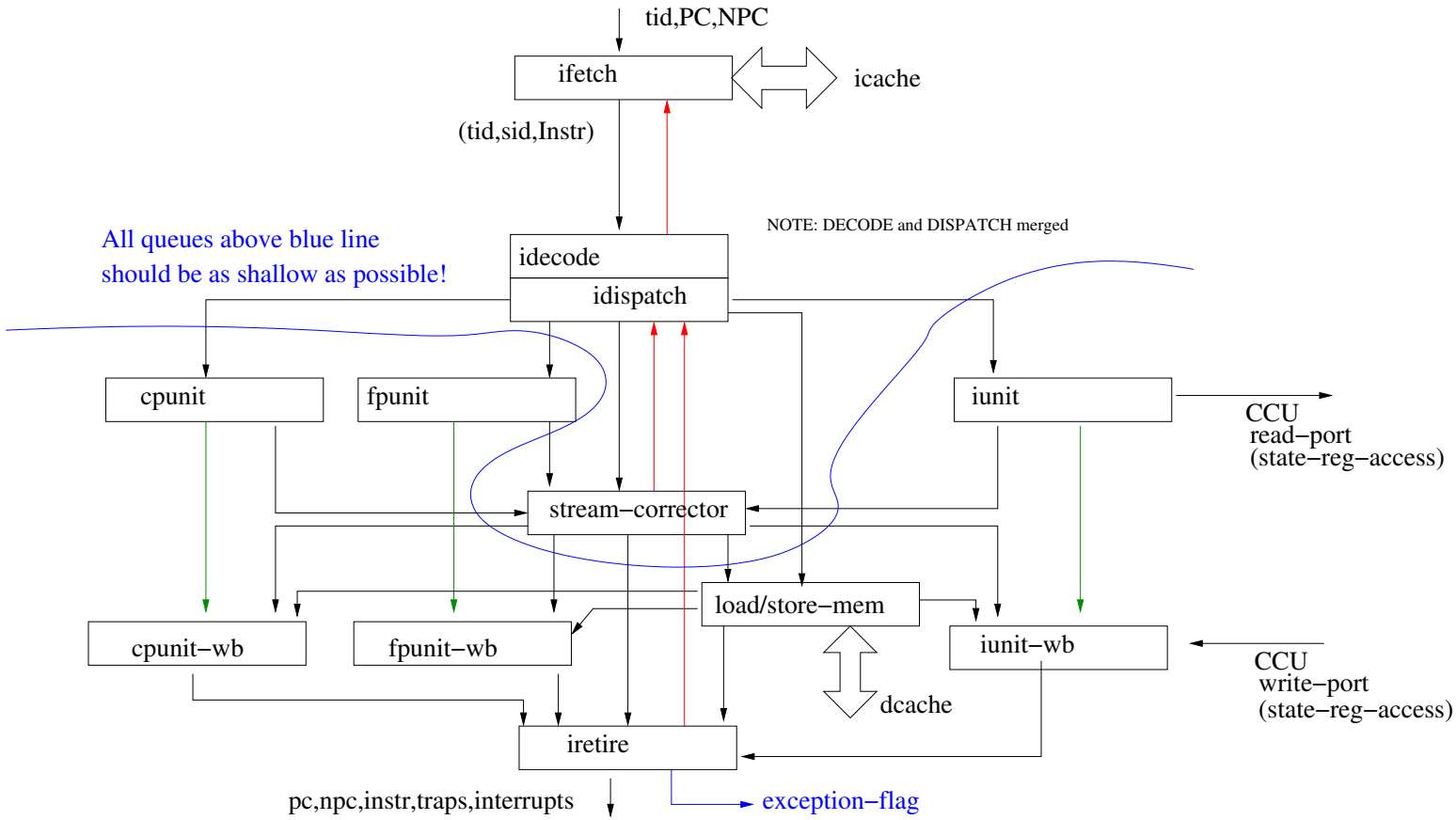
# AJIT processor family components



# Structure of a single AJIT core



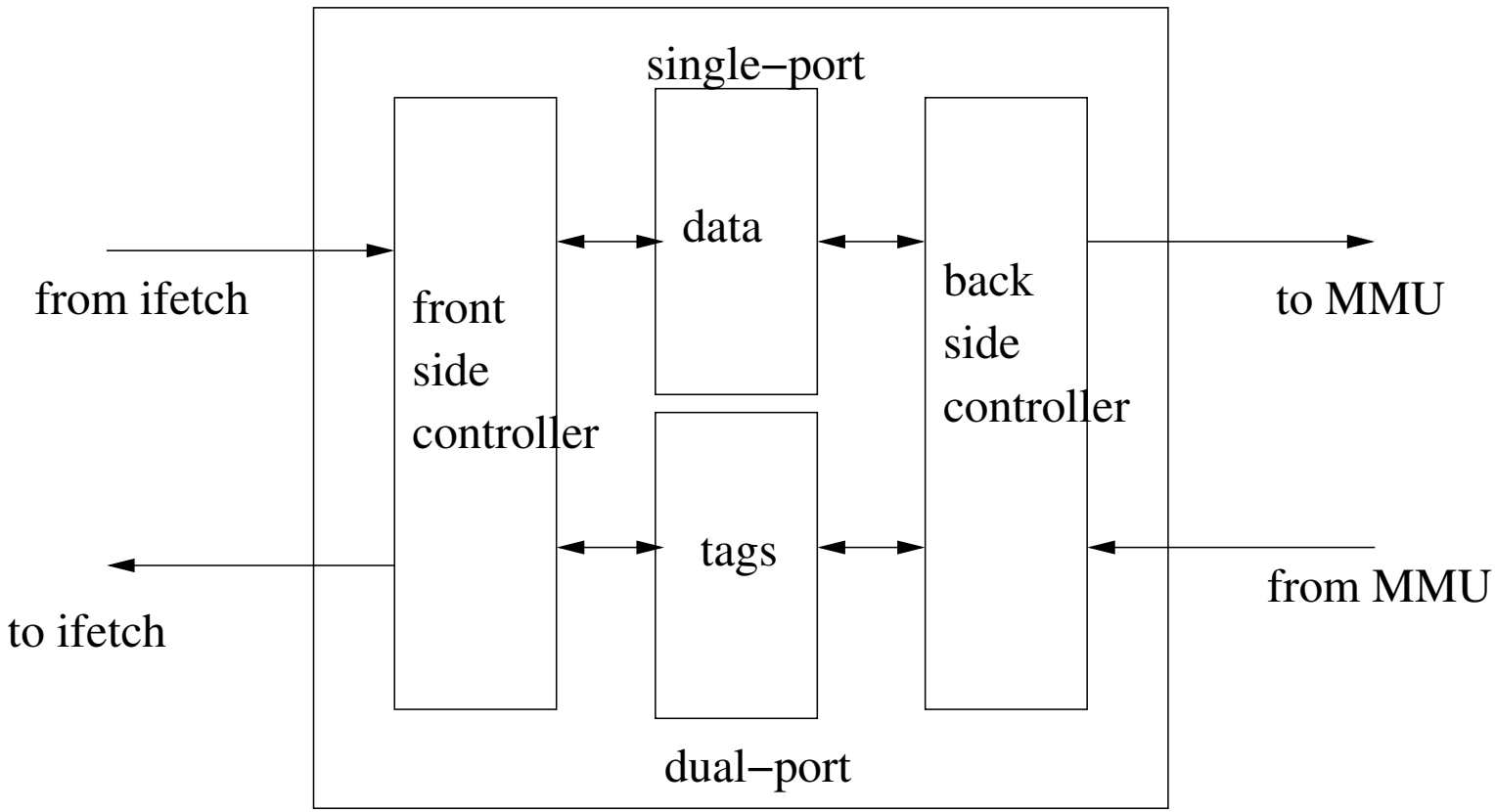
# AJIT thread pipeline



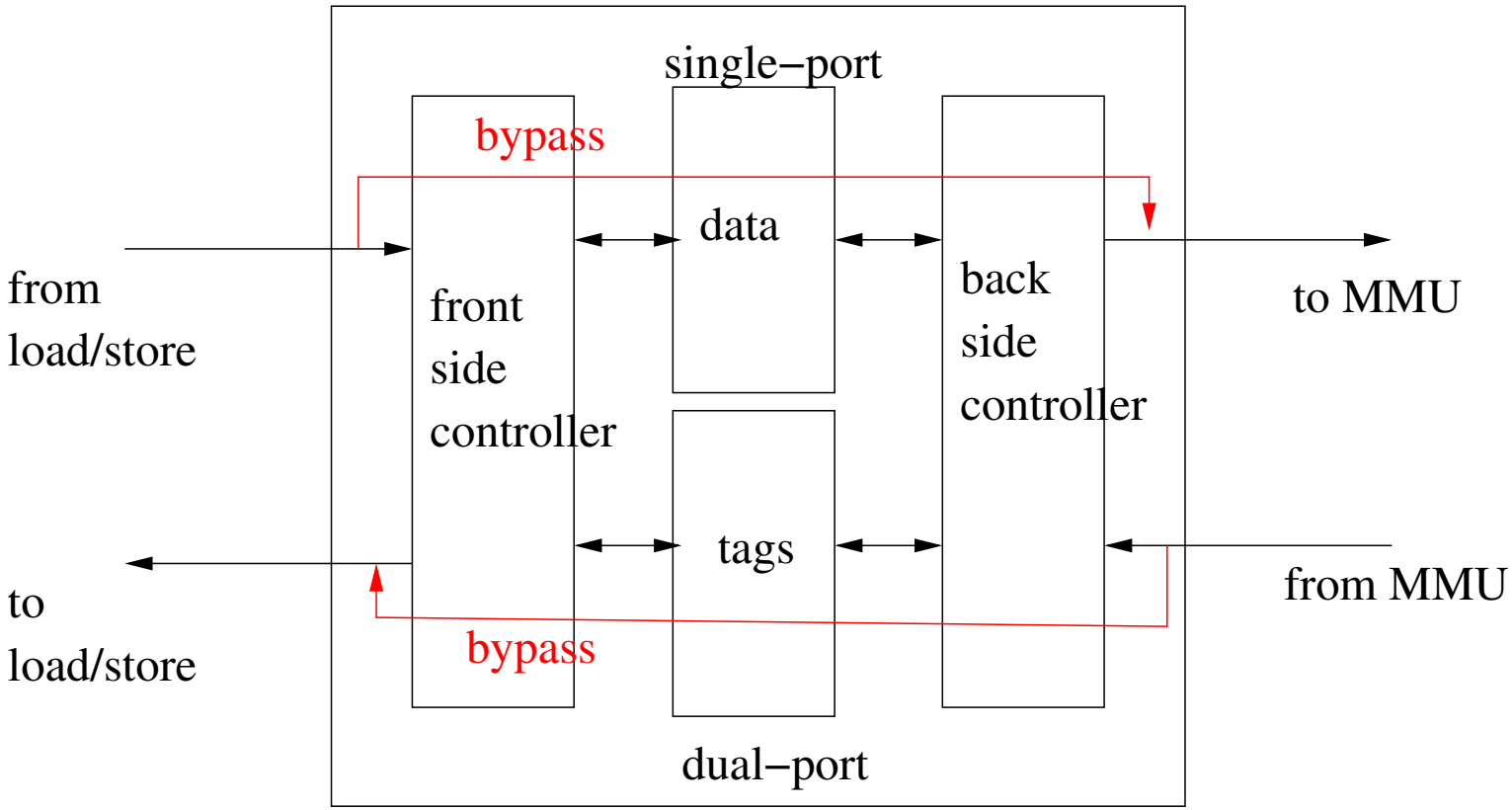
# AJIT Lite processor

- Implements entire SPARC-V8 ISA except for FP and CP instructions.
- Implements 8 register windows.
- Includes caches 8/16/32 KB, up to 4-way set-associative.
- No MMU.
- Full interrupt support.
- Full software-trap support.
- Application areas: industrial controllers, networking, IOT, signal processing, robotics, AI/ML at the edge.

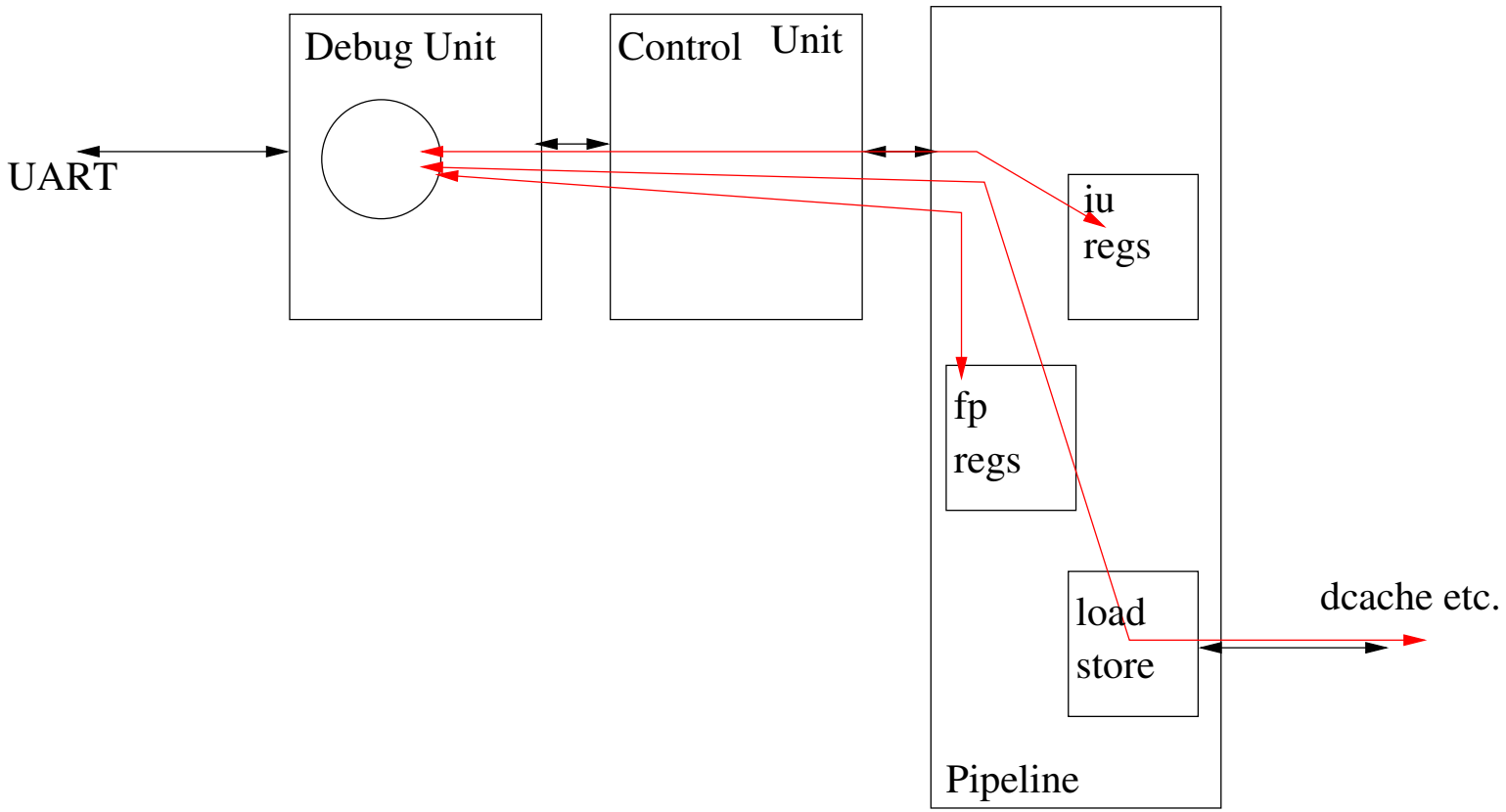
# Instruction cache



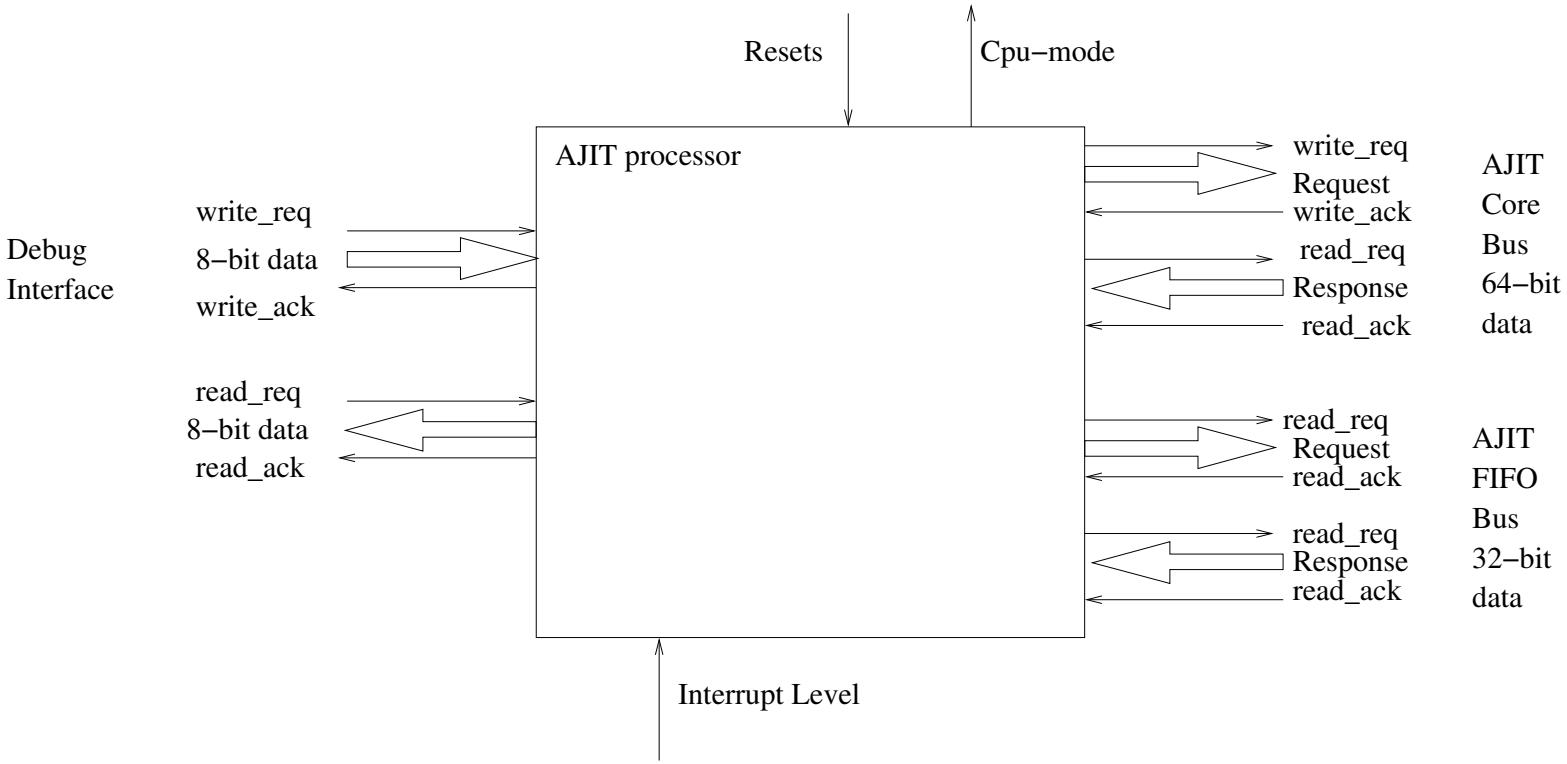
# Data cache


single−port
bypass
data
front
side
controller
back
side
controller
from
load/store
to MMU
tags
to
load/store
from MMU
bypass
dual−port

# AJIT debug monitor



# AJIT processor interfaces



# Ajit-core-bus (ACB) interface

```
-----------------------------------------------------------
Request: 110 bits
Lock    Rwbar    Byte-mask   Addr     Reserved Write-data
-----------------------------------------------------------
(109)   (108)   (107:100) (99:67) (66:64)   (63:0)
-----------------------------------------------------------



-----------------------------------------------------------
Response: 65 bits
Error     Read-data
-----------------------------------------------------------
(64)      (63:0)
-----------------------------------------------------------
```

# Ajit-Fifo-bus (AFB) interface

```
-----------------------------------------------------------
Request: 74 bits
Lock    Rwbar    Byte-mask   Addr     Reserved Write-data
-----------------------------------------------------------
(73)   (72)      (71:68) (67:34)   (33:32)   (31:0)
-----------------------------------------------------------



-----------------------------------------------------------
Response: 33 bits
Error     Read-data
-----------------------------------------------------------
(32)      (31:0)
-----------------------------------------------------------
```

# Ajit-Mini-bus (AMB) interface

- 8-bit request data is organized into a packet, with packet format being

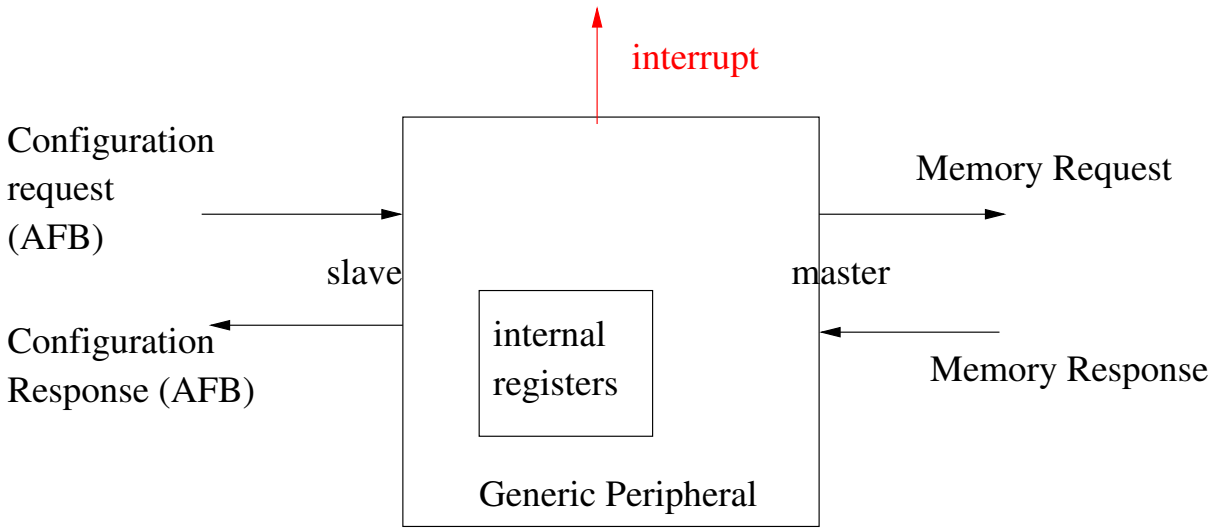
```
-----------------------------------------------
header  a3  a2  a1  a0  b0 b1 ... upto 64 bytes
-----------------------------------------------
```

  The header byte encodes the following fields, and a3, a2, a1, a0 describe the starting byte address, with b0, b1, b3, ... specifying the write bytes.

```
-----------------------------------------------
lock r/wbar  burst-size-in-bytes
-----------------------------------------------
 (7) (6)     (5:0)
-----------------------------------------------
```

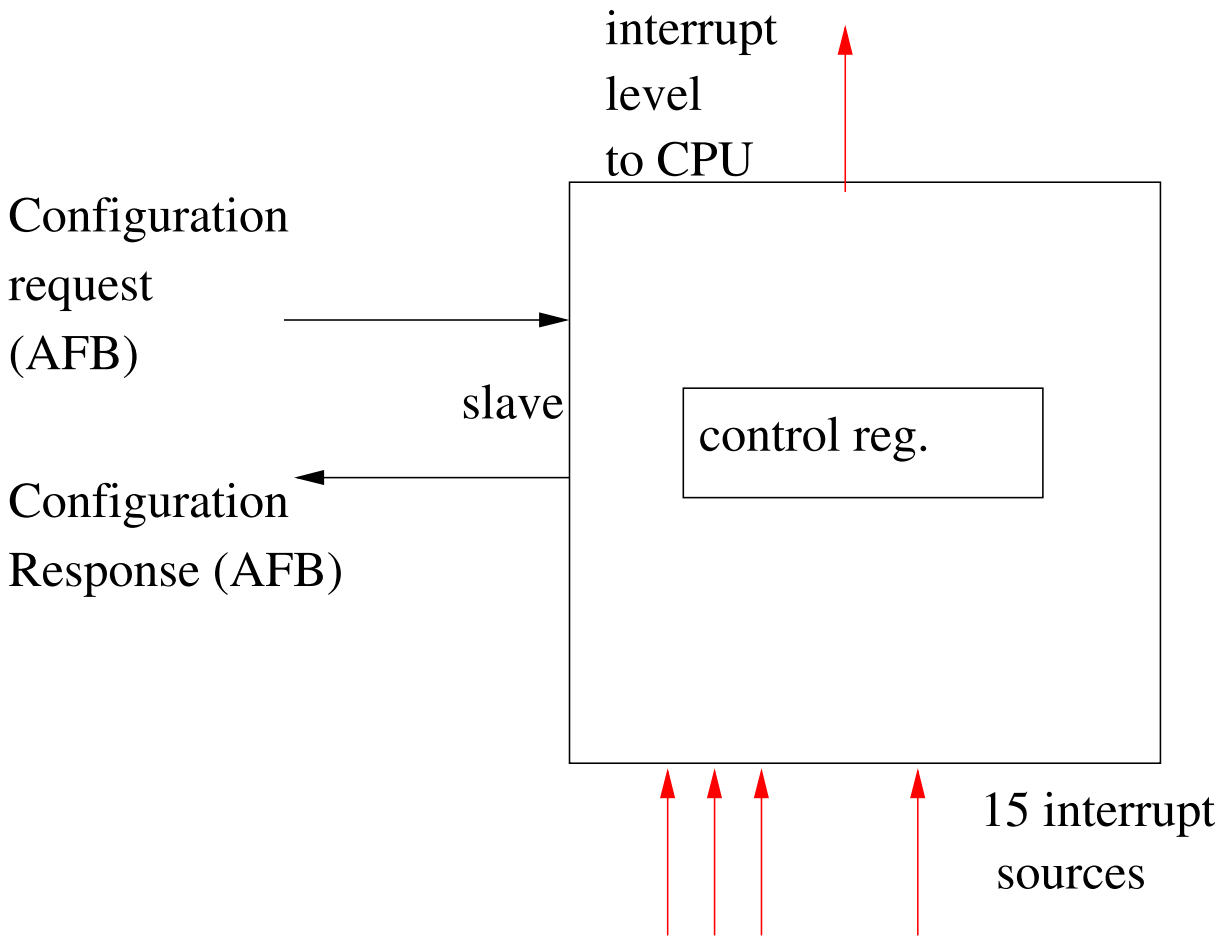
- The response packet consists of up to 63 bytes as specified by the burst size in the header.

# Generic AJIT memory mapped peripheral



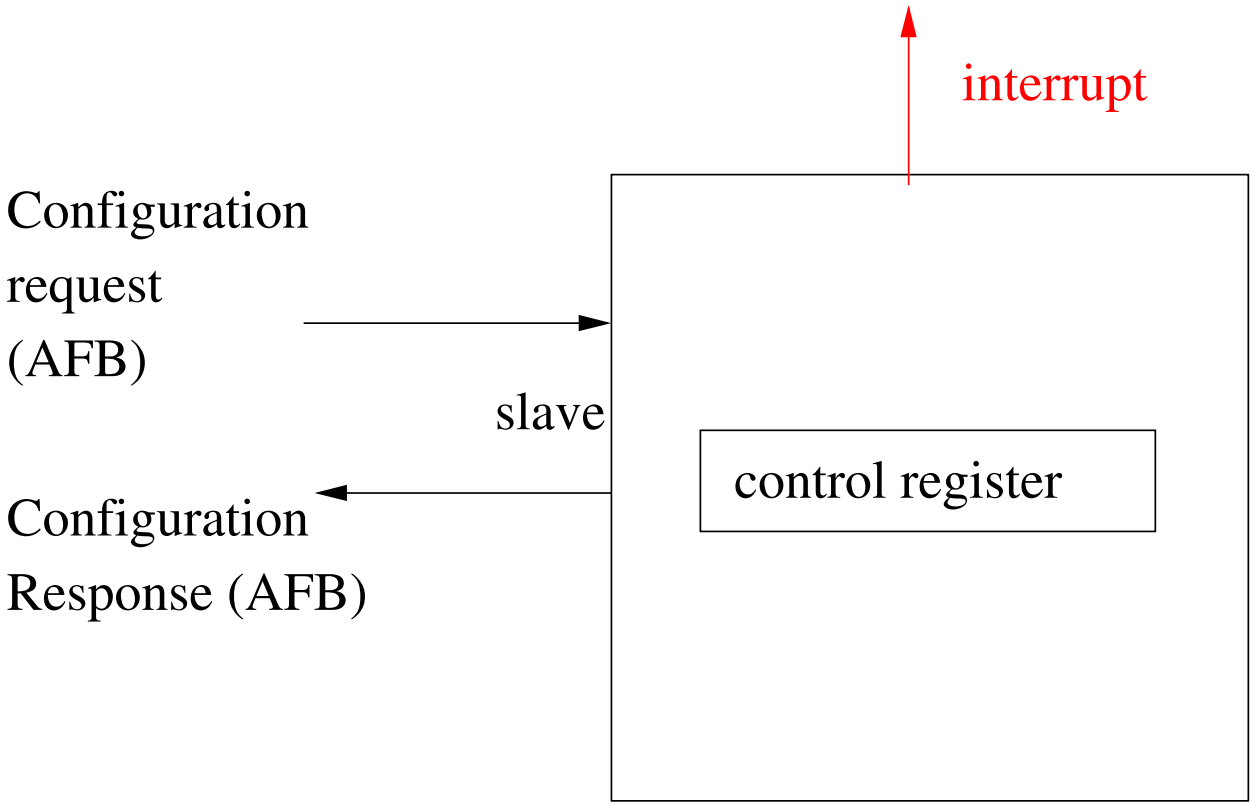
# AJIT interrupt controller

```
control_register
interrupts unused   mask     enabled
 (31:17)    (16)   (15:1)      (0)
```



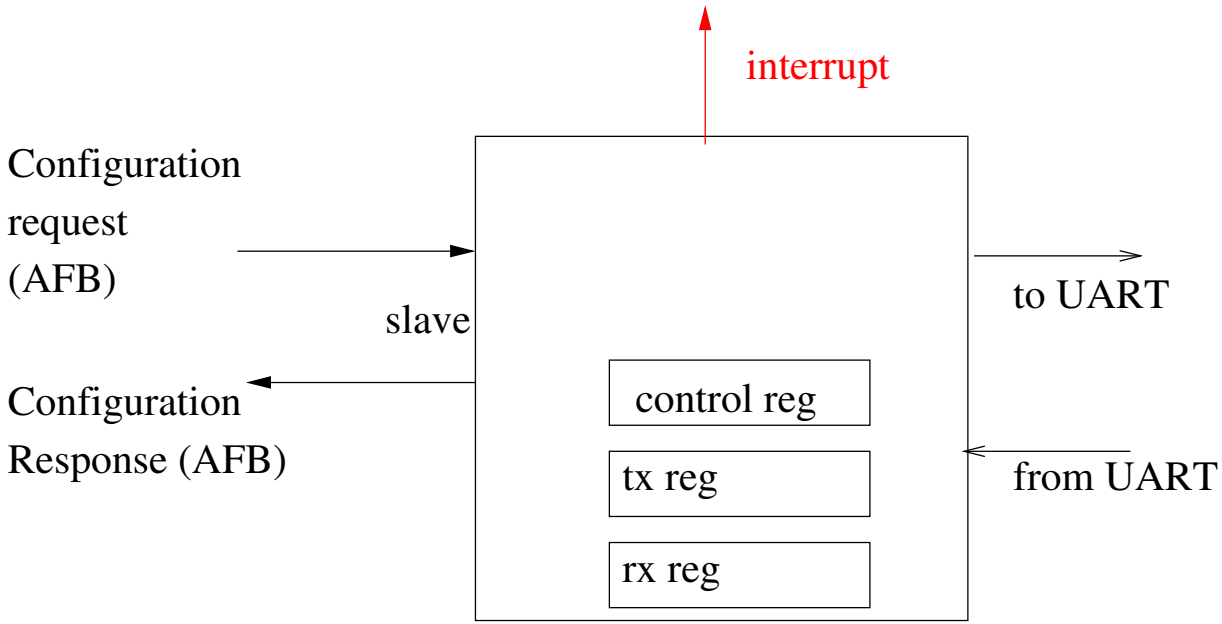
# AJIT timer

```
control_register
timer-count    enable
 (31:1)          (0)
```



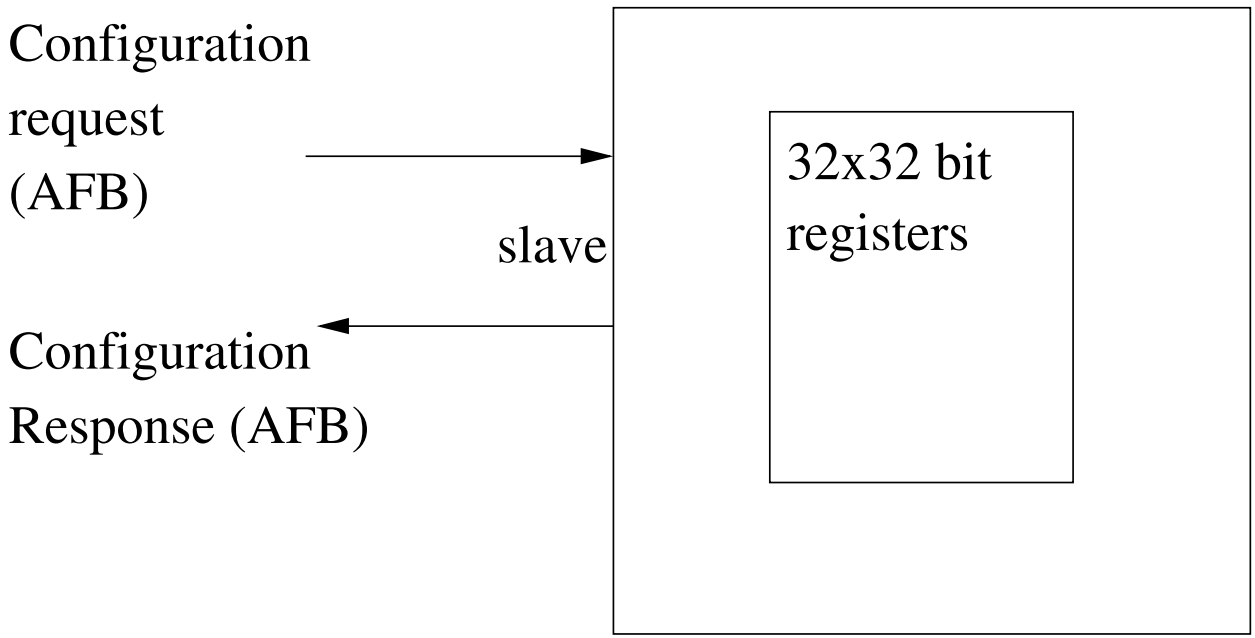
# AJIT serial

Serial device, can be hooked up to UART for now. JTAG is also easy to do.



# AJIT scratch-pad

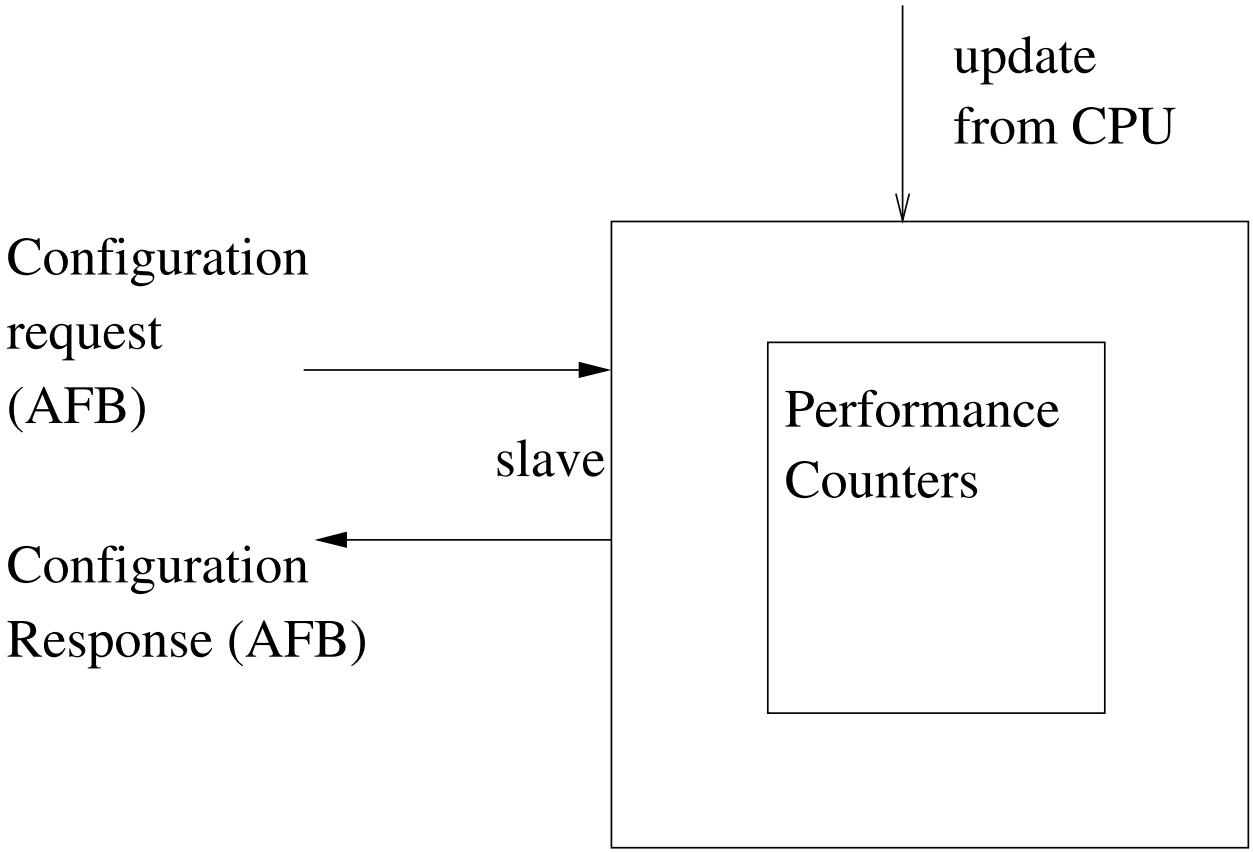
All registers are initialized to 0. Useful for firmware coordination.



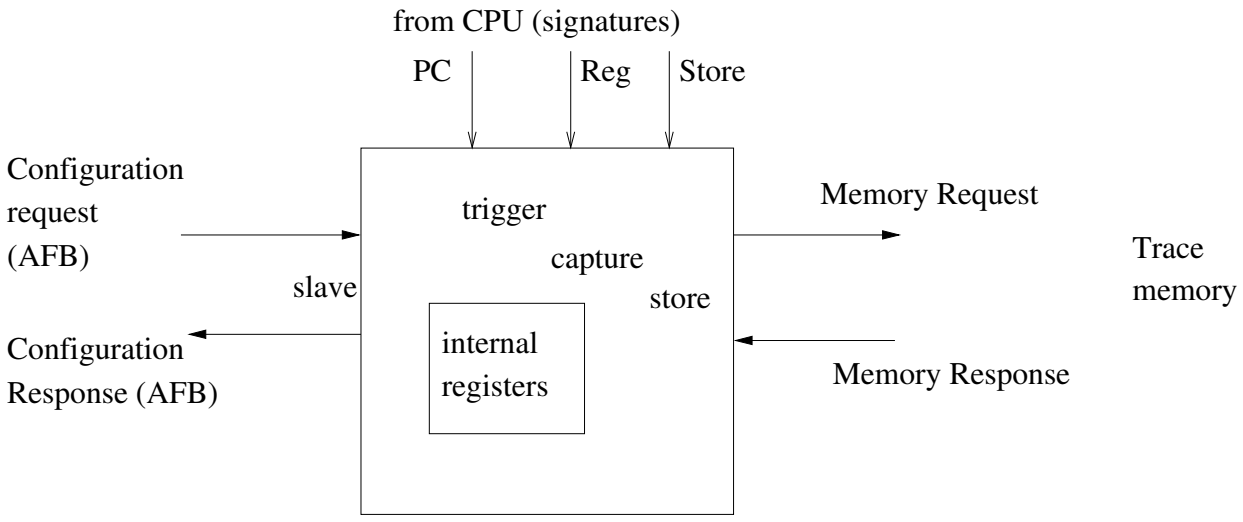
# AJIT SPI, I2C, GPIO peripherals

- SPI master.
- SPI slave.
- I2C master.
- GPIO peripheral.

# AJIT performance counters



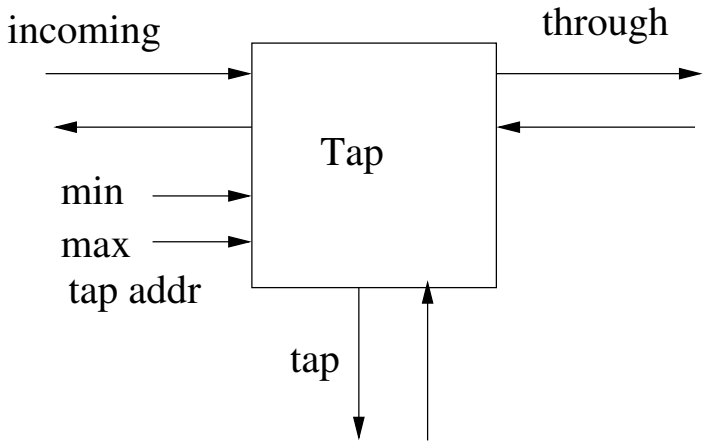
# AJIT trace buffer

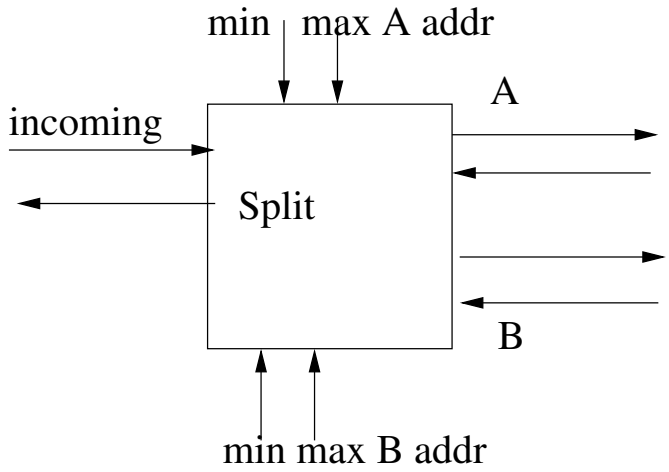


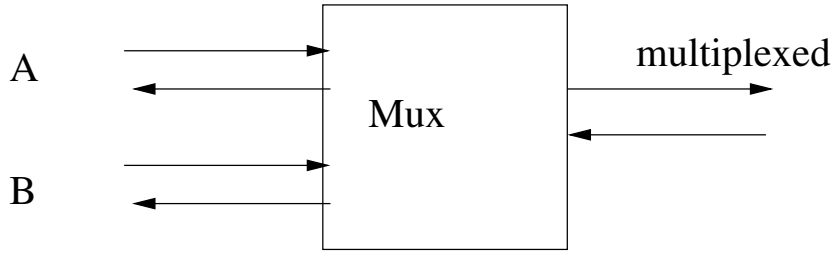
# Network on chip elements

- ACB/AFB bus taps.
- ACB/AFB bus splitters.
- ACB/AFB bus multiplexors.
- ACB/AFB memory.
- L2-cache.
- Coherent memory controller.

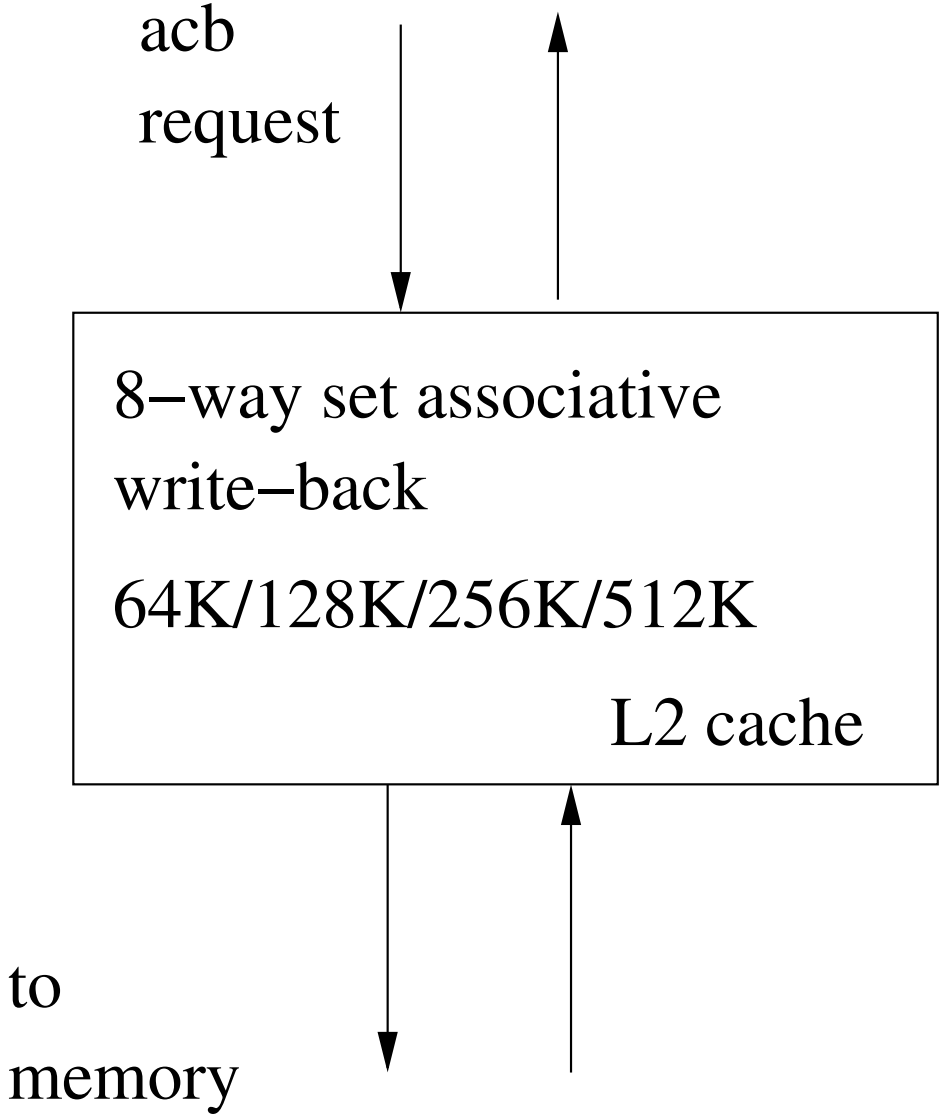
# Tap, splitter, multiplexor



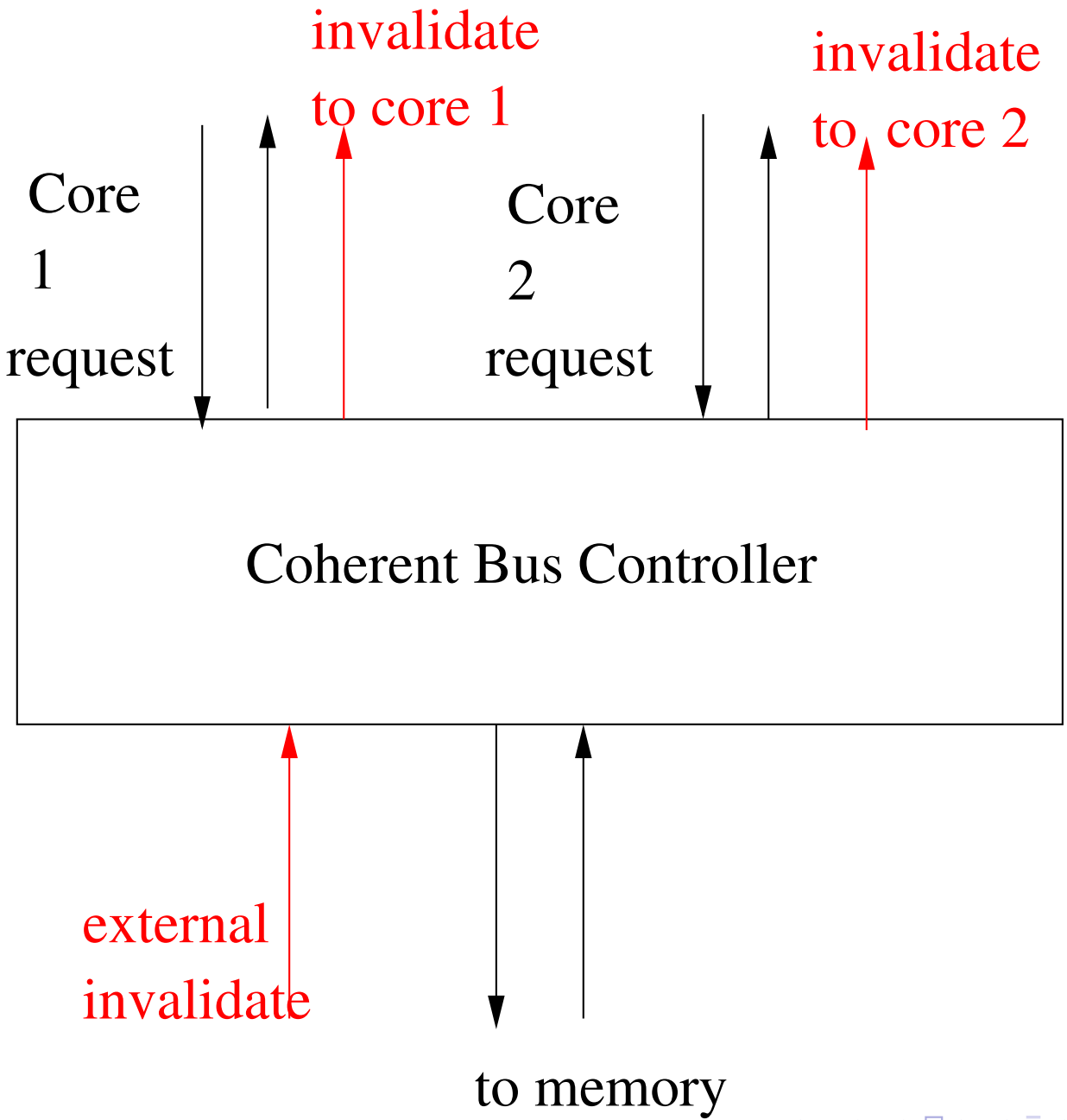



# L2 cache


acb
request
8−way set associative
write−back
64K/128K/256K/512K
L2 cache
to
memory

# Coherent memory controller



# The AJIT tool-chain

- The tool chain:
  - C compiler, Binary utilities, system software support, debug utilities, processor simulator, cooperative RTOS.
  - Processor simulator.
  - Verification test-cases.
  - Application programming helpers.
  - CORTOS2 cooperative RTOS.
  - Lots more.
- Hosted on github.
- The installation and setup process will be described in this section.

# Downloading and installing the tool-chain

- Download the AJIT tool chain from github:

  `git clone https://github.com/adhuliya/ajit-toolchain`

  The "ajit-toolchain" directory will be created in your system. This directory will henceforth be referred to as AJIT_HOME.
- From the AJIT_HOME directory, check out branch "marshal"

  `git checkout marshal`

  You will strictly work in branch marshal.
- Now, download a docker image from the following link.

  `https://drive.google.com/file/d/1xYWJOtoNBamFZHoNgMANXk-9NgOdbU2p/view?usp=sharing`

  The downloaded file will be named

  `ajit_build_dev.tar`
- Continued ....

# Downloading and installing the tool-chain: continued

- If you haven't done so, install docker. If you are running on an Ubuntu distribution, you can do so by navigating to the AJIT_HOME folder, and running the script install_docker.sh. Alternatively, you can directly install docker following appropriate instructions from "www.docker.com".
- Remove any containers that may currently be running:
  `docker rm --force ajit_build_dev`
- Load the tar-file image into Docker:
  `docker load -i ajit_build_dev.tar`
  Note that this has to be done just once.
- Ensure that the image has been loaded correctly. Run
  `docker images ls`
  You should see ajit_build_dev in the list of images.
- Start the ajit_build_dev container. Navigate to the AJIT_HOME/docker/ajit_build_dev directory and
  `./run.sh`
- Continued on the next slide ...

# More on the run.sh command

When you type

```
./run.sh
```

You will start the container. The run.sh command file includes the docker run command

```
docker run \
  --detach \
  --ulimit nofile=100000:100000 \
  --name $_CONT_NAME \
  --mount type=bind,source=$_HOST_MOUNT_DIR, \
      target=$_CONT_MOUNT_POINT \
  $_IMG_NAME;
```

You may wish to mount your local directories in the container.

# More on the run.sh command: mounting your local directories in the container

To mount your local directory in the container, you can use

```
docker run \
  --detach \
  --ulimit nofile=100000:100000 \
  --name $_CONT_NAME \
  --mount type=bind,source=$_HOST_MOUNT_DIR, \
     target=$_CONT_MOUNT_POINT \
  -v /home/madhav/AjitEE:$_CONT_MOUNT_POINT/AjitEE \
  $_IMG_NAME;
```

Note that you should edit the run.sh script as indicated above, and then run the run.sh command.
This will mount the local directory /home/madhav/AjitEE in the container, visible as the directory AjitEE.

# More on the run.sh command: making peripherals visible in the container

- Typically, we will attach two UARTs to the container.
  - The debug UART is used to control the processor using a remote connection,
  - The serial serial UART is used for program I/O.
- Suppose the debug UART is mounted on your local system as /dev/ttyUSB0. In that case, before running the container, set the environment variable

  `export AJIT_DEBUG_UART=/dev/ttyUSB0`
- Suppose the serial UART is mounted on your local system as /dev/ttyUSB1. In that case, before running the container, set the environment variable

  `export AJIT_SERIAL_UART=/dev/ttyUSB1`
- Continued...

# More on the run.sh command: making peripherals visible in the container

- After setting the exports as indicated in the previous slide, the docker run command to be used is

```
docker run \
  --device=$AJIT_DEBUG_UART --device=$AJIT_SERIAL_UART
  --group-add dialout \
  --detach \
  --ulimit nofile=100000:100000 \
  --name $_CONT_NAME \
  --mount type=bind,source=$_HOST_MOUNT_DIR, \
        target=$_CONT_MOUNT_POINT \
  -v /home/user/directory:$_CONT_MOUNT_POINT/directory
  $_IMG_NAME;
```

- A reference script file is provided:
  `docker/ajit_build_dev/run_with_uarts_and_mounts.sh`
  You should edit this file as needed and then run it from the docker/ajit_build_dev directory.

# Installing the tool-chain

- Let us say you have started the container as indicated (with directories mounted, and devices made visible in the container).
- Navigate to the AJIT_HOME/ directory and start a bash shell:

  `./docker/ajit_build_dev/attach_shell.sh`

  This should start bash in the container and give you a bash shell to use the tool chain. The current directory in which you ran docker will be visible in this bash shell.
- You are ready to go. Note that you can create directories and files in the AJIT_HOME directory.

# Potential problems

If the user name is not recognized, go to the /etc/passwords file to find your user-id and use (from the AJIT_HOME directory):

```
docker run -u user-id  -v ... mounts ... \
       --device .... \
       -w $(pwd) -i -t ajit_build_dev  bash
```

If the ajit_build_dev image is not recognized, use

```
docker images
```

to find the image-id of ajit_build_dev and use

```
docker run -u user-id  -v $(pwd):$(pwd)
           -w $(pwd) -i -t  image-id  bash
```

to start the shell in the container.

# Build the tool-chain

Confirm that you are in the shell attached to the Docker container.

- In the bash shell that you have attached to the container, run

  ```
  source set_ajit_home
  source ajit_env
  ```

- And then

  ```
  ./setup.sh
  ```

  After some time, if all goes well, your tool-chain will be ready. If the tool-chain has been built already, you can skip this step (which takes a while to finish).

- Documentation on the AJIT processor and environment around it is provided in the docs/processor/ directory in AJIT_HOME.

# Contents of the tool-chain

- Compiler (gcc, g++), UCLIBC.
- GNU utilities: debugger, assembler, dis-assemblers, linker.
- AJIT hardware access routines.
- CORTOS2 cooperative RTOS.
- Trap handlers, interrupt handlers.
- MMAP routines.
- AJIT IPI routines.
- AJIT system-call interface routines.
- AJIT processor reference simulator.
- Verification tests.
- Documentation

# Summary

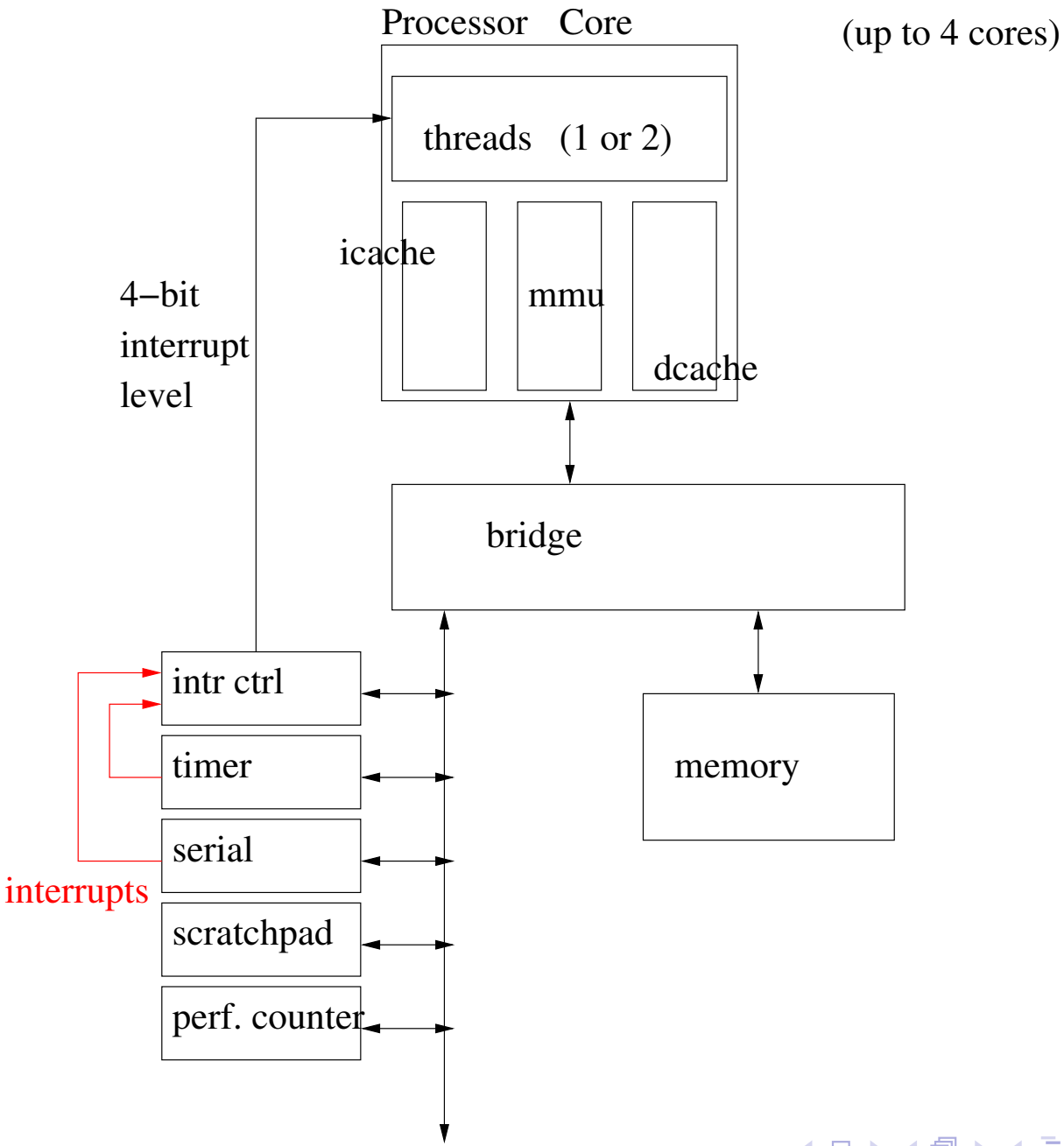
- Clone the AJIT tool-chain.
- Check out branch marshal.
- Get the ajit_build_dev docker image.
- Run the docker container (attache devices, host directories).
- Attach shell to the running docker container.
- If not already done, build and set up the tool-chain.
- You are ready to go.

# AJIT C-model

The AJIT C-model implements a software model of a processor platform with

- Up to 4 cores, each with 1 or 2 threads.
- Peripherals: serial devices (2), timer, interrupt controller, scratch-pad, performance counters.
- Main memory.

# Platform modeled by AJIT C-model



# The AJIT C-model

The following command

```
ajit_C_system_model -m add_test.mmap -i 0x40000000
```

loads memory map file into memory, and executes starting from address 0x40000000.
For all the options, type

```
ajit_C_system_model -h
```

We will describe these options in detail later.

# Compiling and running a program

Start with this program (lets say it is in file prog.s):

```
.section .text.ajitstart
_start:
mov 0x1, %g1
mov 0x2, %g2
add %g1, %g2, %g3
ta 0
```

# Compiling and running the assembly program

We use the following compilation script from the BIGMEM directory:

```
... some stuff omitted ...
TEXTBASE=0x0      # location of
                  # first instruction to be executed
DATABASE=0x1000   # location of data section
#1 generate linker script
makeLinkerScript.py -t $TEXTBASE -d $DATABASE \
              -o customLinkerScript.lnk
#2 compile the application
compileToSparcUclibc.py -N prog -s ./prog.s
```

# Compile and run

Run

```
./compile_for_ajit_uclibc.sh
```

This produces a file prog.mmap. This is a memory image of the program mapped to physical memory. Run

```
ajit_C_system_model \
            -m prog.mmap \
            -i 0x0
```

Lots of useful information is printed to the terminal in which you are running the C model.

# Checking your results

We want to check if the post-condition of the simple assembly program is $< g1 >= 1$, $< g2 >= 2$, $< g3 >= 3$.
Create a results file with the contents

```
g1=0x1
g2=0x2
g3=0x3
```

and run

```
ajit_C_system_model \
            -m prog.mmap \
            -i 0x0 \
            -d -r prog.results -l prog.log
```

The simulator will check the post condition and put the results of the check into the "prog.log" file.

# More examples later

There will be more examples later..

# AJIT C Model: options

```
-n <number-of-cores>
-t <number-of-threads-per-core>
-m <mmap-file>
-D <dcache-size-in-lines>  (optional, default=512)
-N <dcache-size-in-lines>  (optional, default=512)
-A <dcache-associativity-in-lines>, optional  (default=1)
-Q <icache-associativity-in-lines>, optional  (default=1)
-g   (optional, use to connect debugger)
-p   <port-number>   (debugger tcp port, 8888 typical)
-d   (optional, specify to check post-condition0
-r  <post-condition-file>
-l  <post-condition-check-results>
-w   (optional, use to dump register write, store traces)
-i  <start-pc>  (optional, default=0x0)
-R  <rng-seed>  (optional, initialize memory with random v
```

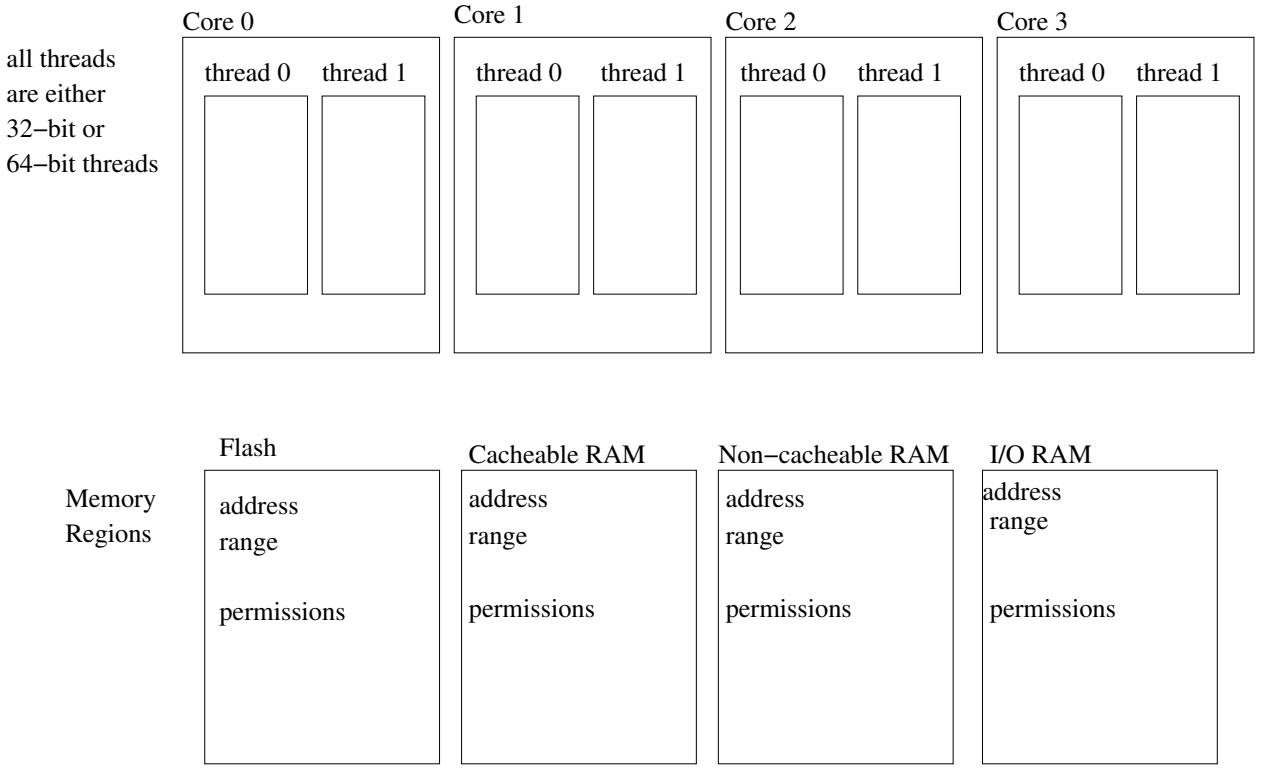
# Summary

- Download and installation of AJIT tool chain.
- First look at the AJIT processor.
- Description of AJIT C model.
- Compiling and running a simple assembly program on the C model.

# CORTOS2: cooperative RTOS infrastructure

- Developed at IIT Bombay (Anshuman Dhuliya).
- Simplifies application development for AJIT processor based systems.
- Using CORTOS2:
  - Describe the AJIT processor based system: number of cores, number of threads per core, memory regions, memory mapped I/O.
  - Describe the application: initialization code, start routine for each thread, source files, compiler options.
  - Specify the exception handlers: Interrupt handlers for interrupts of interest, software trap handlers.
  - Set up run time logging, debug options.
- CORTOS2 provides useful libraries for locks, queues, dynamic memory allocation.
- Several application development support libraries are available: printing, timing, co-routines, co-operative tasks.
- All this information is in a single YAML file.

# CORTOS2: generic system view




Software traps: handlers
interrupt servicing: Interrupt−level <−> Interrupt−handler

Start routine for each thread

Initialization functions etc.

number of lock, queue resources..

stack for each thread.

# Using CORTOS2

- In the working directory, create the following files: config.yaml, a description of the processor system, build.sh (a script to build), and your source code. The build.sh file contains

  ```
  #!/usr/bin/env bash
  cortos2 build "$@";
  ```

- Fill the config.yaml file to describe the system configuration, the build options (add additional source files if needed).
- Use build.sh to compile the program. This produces a memory map in the

  `cortos_build/main.mmap.remapped`

  file.
- Run the memory map using the C model or actual hardware.

# Describing the processor in the config.yaml file

```
Processor:
  # Required: Number of cores and threads per core.
  Cores: 1
  ThreadsPerCore: 1
  ISA: 32 # 32/64 bit (Default: 32)
  MMU: No
  FPU: No
```

# Description of the memory configuration

```
Memory:
  MaxPhysicalAddrBitWidth: 36
  Flash:
    StartAddr: 0x0  # physical address
    SizeInMegaBytes: 16
    Permissions: RXC  # (Read,Write,eXecute,Cacheable)
  RAM:
    StartAddr: 0x40000000
    SizeInKiloBytes: 1024
    Permissions: RWXC
  NCRAM:
    StartAddr: 0x40100000
    SizeInKiloBytes:   1024
    Permissions: RWX
  MMIO: # Memory Mapped IO
    StartAddr: 0xFFFF0000  # physical address
    EndAddr:   0xFFFFFFFF
    Permissions: RW
```

# Text and data sections

- For the application, CORTOS2 starts the text section at the start address of RAM.
- The data section is adjusted based on the size of the text section.
- Stacks are allocated for each thread, based on the programmer's directive in the configuration YAML file.
- User can focus on functionality and need not worry about initialization details (set up of run time, virtual memory mapping etc.).

# CORTOS2 system configuration: software build options

```
Software:
  BuildAndExecute:
    LogLevel: DEBUG
    OptimizationLevel: 2 # i.e. O2
    EnableSerial: Yes
    Debug: No
    FirstDebugPort: 8888
    # specific to applications..  arguments to  compiler.
    BuildArgs: '-D CLK_FREQUENCY=80000000 '
               ' -D NREPS=256 '
               ' -D NCRAM_BASE=0x40100000'
```

# CORTOS2 system configuration: attaching functions to and allocate stack for threads

```
# Required: Which functions execute on each ajit thread.
ProgramThreads:
  - CortosInitCalls:
      - main_00
    StackSize:
      SizeInKiloBytes: 8
```

# CORTOS2 system configuration: dynamic memory, locks, queues

```
DynamicMemory: # Dynamic Memory Configuration.
  # SizeInBytes: ...
  SizeInKiloBytes: 1000
  # SizeInMegaBytes: ...

Locks: # Locks for synchronization between threads.
    Cacheable: 32
    NonCacheable: 32
```

# API

```
cortos_reserveQueue
cortos_freeQueue
cortos_writeMessages
cortos_readMessages

cortos_bget
cortos_bget_ncram
cortos_brel
cortos_brel_ncram
```

```
cortos_reserveLock
cortos_freeLock
cortos_lock_acquire
cortos_lock_acquire_buzy
cortos_lock_release

etc....
```

# CORTOS2 sample application: hello_world

```
#include <ajit_access_routines.h>
#include <cortos.h>
int main() {
    cortos_printf("hello world.\n");
    return(0);
}
```

# System Software resources: overview

- Processor observation and control resources.
  - AJIT access routines.
- Virtual memory management.
- Interrupt management.
- Software trap management.
- Device addresses.

# System Software resources: ajit_access_routines

In directory
tool-chain/AjitPublicResources/tools/ajit_access_routines_mt
include/ and src/ directories.

```
#include <ajit_access_routines.h>
```

# System Software resources: important AJIT access routines

- Set and get register values.
- Access memory using bypass (accessing physical memory).
- Read processor clock, descriptor (cache-size etc.).
- Peripheral access: interrupt controller, serial devices, scratch-pad etc.
- Device addresses.
- Take a look.

# System Software resources: virtual-physical mapping

In directory
tool-chain/AjitPublicResources/tools/ajit_access_routines_mt
include/ and src/ directories.

```
#include <ajit_mmap.h>
```

# System Software resources: virtual-physical mapping

- Allocation and management of page table physical memory.
- Page table traversal.
- Adding a virtual-physical page table entry.
- Removing a virtual-physical page table entry.
- Doing a virtual to physical lookup.
- Take a look.

# System Software resources: Interrupt management

In directory
tool-chain/AjitPublicResources/tools/ajit_access_routines_mt
include/ and src/ directories.

```
#include <ajit_mt_irc.h>
#include <ajit_ipi.h>
```

- Enable/disable interrupt controller.
- Mask/unmask interrupt (on a per CPU basis).
- Inter-processor interrupt mechanism.
- Take a look.

# System Software resources: Software trap management

In directory
tool-chain/AjitPublicResources/tools/ajit_access_routines_mt
include/ and src/ directories.

```
#include <ajit_mt_sw_traps.h>
```

- assign vectored software trap handler.
- easy to do with CORTOS2.

# Application Software resources: overview

- ajit_context_optimized: get/set/swap context, coroutines, cooperative tasks (threads).
- protothreads: lightweight stack-less threads.
- athreads, thread_channel: multi-thread workload creation and management.
- bscanf: string input (like scanf).
- rng_marsaglia: pseudo random number generator.

# Application Software resources: overview

- Context management: get/set/swap context.
- Coroutines.
- proto-threads, athreads, thread-channels.
- Tasks and task-scheduler.
- Especially useful in multi-threaded, multi-core AJIT processors.

# Concepts covered

- The SPARC V8 ISA.
  - registers, memory, instructions, exceptions.
- The AJIT processor family.
  - The AJIT CPU, caches, peripherals.
- The AJIT tool-chain.
  - Download, setup.
- The AJIT processor C-model (simulator).
- The CORTOS2 co-operative operating system.
- Software resources.

# Moving on

- A look at the AJIT processor RTL.
- Support VHDL libraries.
- The FPGA prototype.
- Code examples.
  - Assembly.
  - Bare metal C.
  - CORTOS2.