Using the AJIT processor: a short course

Madhav Desai
Department of Electrical Engineering
IIT Bombay

January 13, 2024

Getting started

- The AJIT single-core, single-threaded processor.
- The tool chain: compilers, binary utilities, support software.
  - Installation and setup.
- The AJIT processor simulator.
- Compile and run a program on the simulator.

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:
  - The processor includes a floating point unit, 16KB Data/Instruction caches, a memory management unit and various peripherals.
- 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 supported.

Single core single thread AJIT processor core

- IEEE 1754 standard ISA (Sparc V8).
- IEEE floating point unit.
- VIVT ICACHE (32kB) DCACHE (32kB) with 2-cycle latency on hit, MMU with 256 entry TLB and hardware page table walk.
- Single issue, in-order, seven stage one instruction-per-cycle instruction pipeline, with branch-predictor.
- C compiler, debugger, Linux.

Downloading and installing the tool-chain

The tool chain can be downloaded from github using

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

Now, download the following docker image

```
https://docs.google.com/uc?export=download&id=
                1ck2jSq8LTO-q2RFrwc49mPkXTH9LecVx
```

The downloaded file will be named

```
ajit_build_dev.tar
```

Downloading and installing the tool-chain

Remove any containers that may be running

```
docker rm --force ajit_build_dev
```

Load the tar-file into Docker:

```
docker load -i ajit_build_dev.tar
```

Navigate to your home directory and start a container with attached bash shell:

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

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.

Potential problems

If the user name is not recognized, go to the /etc/passwords file to find your user-id and use

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

If ajit_build_dev 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
```

Build the tool-chain

Navigate to the ajit-tool-chain directory that you have cloned into.
Run

```
source set_ajit_home
source ajit_env
```

and then (if setup has not been done earlier)

```
./setup.sh
```

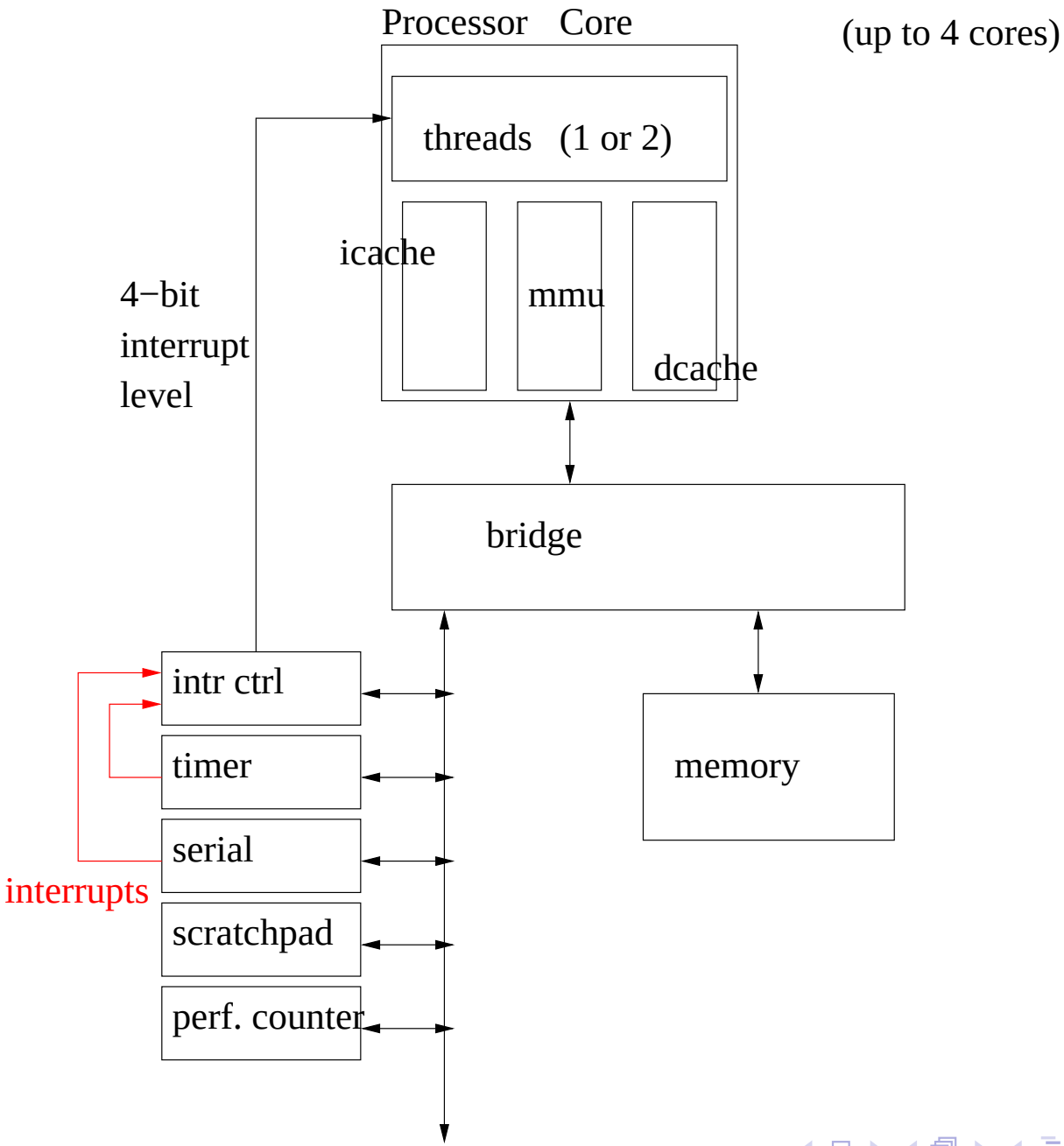
After some time, your tool-chain will be ready.
If the setup has been done earlier, you can skip this step (which takes a while to finish).
Documentation on the AJIT processor and environment around it is provided in the docs/processor/ directory.

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

Go to the examples/ directory which has been shared with you. In this directory, go to:

```
examples/mp_printf
```

The directory contains the following:

```
init.s      initialization code to
            set up stack etc.
main.c      the C routine
BIGMEM/VMAP.TXT
            a virtual to physical
            mapping of pages.
BIGMEM/compile_for_ajit_uclibc.sh
            compile directory
```

Compiling and running a program: init.s

```
.section .text.ajitstart
.global _start;
_start:
set 0xfff01ff8, %sp ! stack pointer
        set 0xfff01f00, %fp ! frame pointer
... set up wim, tbr registers.....
        ... set up virtual->physical mapping...
set 0x10E7,%l0
wr %l0,%psr    // psr.

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

call main // jump to main
nop
ta 0 // halt on return.
```

Compiling and running a program: main.c

- The main program does a series of prints. Note the included directories!
- The VMAP.TXT file defines a virtual to physical mapping. For example, the line

  `0x0 0x40000000 0x40000000 0x2 0x1 0x3`

  means, "for context 0, map virtual page at address 0x40000000 to physical page at address 0x40000000 (page size is 256KB), and the page is cacheable, with access permissions 0x3 (supervisor and user can read/write execute)". For more documentation, read the processor description in AJIT processor description document for details about VMAP files.
- The compile script uses a Python script, described next.

Compilation script

```
... some stuff omitted ...
TEXTBASE=0x40000000  # location of
                     # first instruction to be executed
DATABASE=0x40040000  # location of data
VMAP=VMAP.TXT
CLKFREQ=80000000
#1 generate linker script
makeLinkerScript.py -t $TEXTBASE -d $DATABASE -o customLink
#2 compile the application (run with -h to get options).
compileToSparcUclibc.py .... lots of options ....
                     (use -h flag to see all options)
```

Compile and run

Go to the BIGMEM directory and run

```
./compile_for_ajit_uclibc.sh
```

This produces a file printf_test.mmap.remapped in the BIGMEM directory. This is a memory image of the program mapped to physical memory. Run

```
ajit_C_system_model \
            -m printf_test.mmap.remapped  \
            -i 0x40000000
```

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

Summary

- Download and installation of AJIT tool chain.
- Description of AJIT C model.
- Compiling and running a simple program.

CORTOS2 (Pushkar): 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: An AJIT processor based system

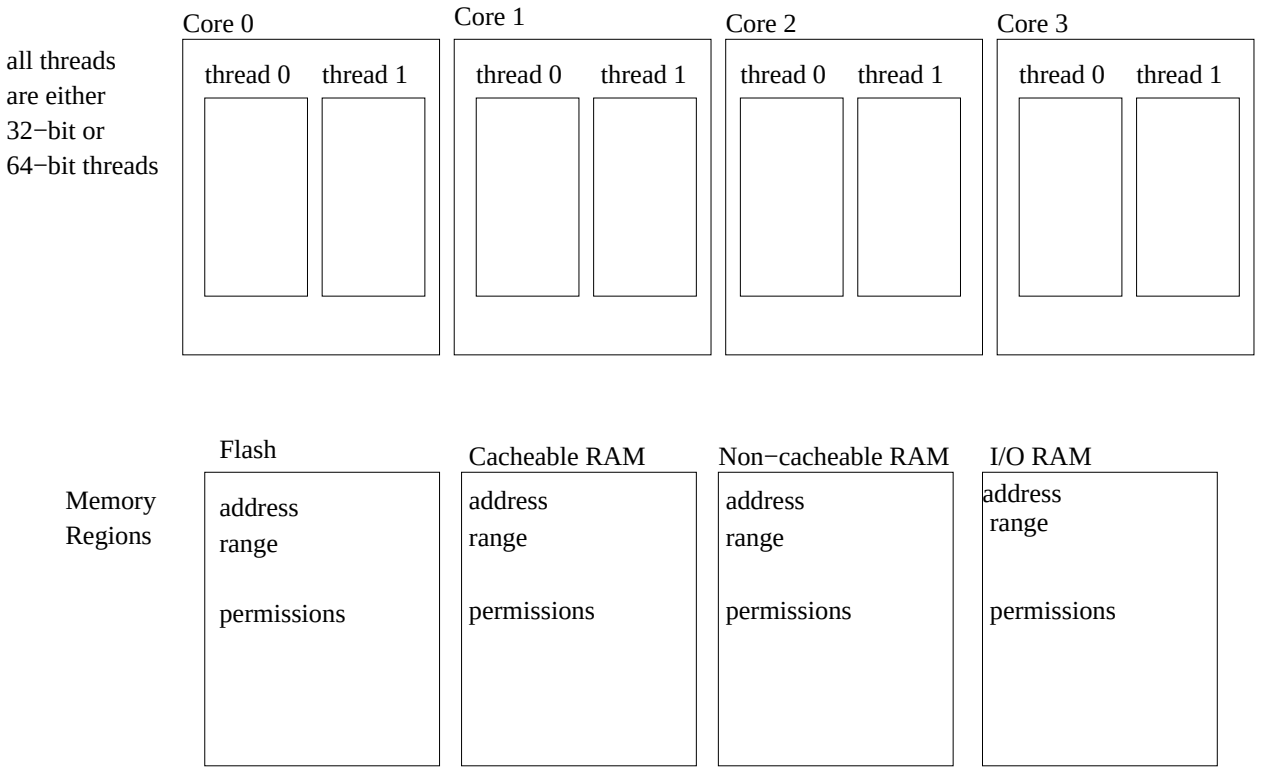


Figure: System configuration template in CORTOS2

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.

CORTOS2: describing the system 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)
```

CORTOS2: describe 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
```

CORTOS2 system configuration: 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

DynamicMemory: # Dynamic Memory Configuration.
  SizeInKiloBytes: 1000

Locks: # Number of locks available to user.
    Cacheable: 32
```

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.

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.

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.

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.

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.