COP 701(Summer 2015)

ASSIGNMENT 1

ELF Parser
ARMv8 Timing/Energy Simulator
Debugger

In this assignment you will develop/update an ELF Parser and ARMv8 instruction set timing/energy simulator. You will start with an ELF format binary compiled for the ARM architecture. The instructions appearing in the binary will be a subset of the ARMv8 instruction set. The ELF Parser will have to parse the ELF format binary, extract the instructions (along with other information), and serve as input to ARMv8 instruction set simulator. Your simulator is expected to model a 5 stage single issue pipeline[5] and should keep track of the state of the memory and all the architectural, pipeline registers. Along with the simulator you are expected to design/update a tiny debugger.

To help you start quickly, we provide you with a basic ARM-v8 simulator[1] (coded in python). The given simulator is equipped with very basic ARM-v8 instructions, along with an ELF parser and a tiny debugger. A list of implemented instructions is present in the **Instruction List Section**. So you are required to:

Add more instructions to the simulator.

Model a 5-stage pipeline.

Add support for energy and timing simulation and optimization.

The assignment will have 3 phases and has to be done in groups of **two.** The preferable language is python. In all phases you required to do regular commits to the BitBucket repository.

PHASE 1 : SIMPLE INSTRUCTIONS

[Deadline: Aug 9 2015 23:55 PM]

In the first phase you are expected to model the five stage pipeline. You may start by understanding the provided code[1]. At the end of this phase you should have added more instructions to the simulator & debugger. You are expected to handle previously coded instructions and new instructions mentioned in **Instruction List Section**. Assume Single cycle delay for all Function units(Caches, RF, ALU) and forwarding paths available. However, modelling of Mem-to-Reg data hazards should be done. Assume RF has 2 Read ports and 1 Write post and memory has 1 Read and 1 Write port. In the debugger, functionality such as printing contents of registers, number of cycles program ran, number of stall cycles etc. should be added.

PHASE 2 : COMPLEX INSTRUCTIONS AND TIMING

[Deadline: Aug 17 2015 23:55 PM]

The simulator should support all the instructions(**Instruction List Section**). The instruction set also includes floating point instructions. Your simulator must model multi-cycle functional units and data hazards so that we can obtain total number of execution cycles for the program and performance(activity) counters values for each functional unit. To obtain the latency and energy numbers for each functional unit, you need to parse config.xml[7] .The debugger must be updated in accordance with the pipelined execution of instructions.

PHASE 3 : ENERGY ESTIMATION AND EDP OPTIMIZATION

[Deadline: Aug 27 2015 23:55 PM]

ARMv8 simulator should support total energy estimation. We will provide you multiple frequency points you should be able to estimate the voltage of operation and the energy, power numbers. Finally, you will be required to graphically represent the variation of Energy Delay product with voltage. See **energy modelling section** for more details.

It is essential to start early

We have provided an initial code[1] to help you start. Please go through it. Read the ELF parser. Play with the debugger. Check the code flow. Understand the three stage pipeline.

Submissions for each phase must be made on Moodle as well as BitBucket before the respective deadlines. See the submissions section for more detail.

Refer to the ARMv8 reference manual[2] for information on the instruction set and the encoding for the different instructions.

A list of instructions that will appear in the binary will be provided in a separate file. We will also provide you with a few binaries and their disassembled output.

You will want to generate your own test cases. For this you will need to setup a cross compiler. Refer to the cross compiler section for this.

More detailed notes on how the debugger should function can be found in the debugger section.

It is recommended to use git for local development.

In the five stage pipe-lining you can assume: There is no need to model Branch Predictors. Also a file will provided to you indicating the energy and delay numbers for each unit. Please see pipe-lining section for details.

Please join Piazza. It will be our discussion forum for the rest of the course. No queries will be responded to on mail.

Academic Integrity Code

Academic honesty is required in all your work. Discussion of assignments on Piazza is fine(TAs will monitor Piazza) but looking at someone else's work and then doing your is not. You must do all written and programming assignments on your own, except where group work is explicitly authorised. If you use parts of a solution or code from other sources (such as Internet etc.), you should explicitly mention it in your submission. Letting your work become available or visible to others is also cheating. The first instance of cheating will straight-away invite an 'F' grade in the course and a referral to the disciplinary committee.

In the past years disciplinary action has been taken on students found copying.

Moss (for a Measure Of Software Similarity) will be run and your code will be matched to check similarity with this year and previous years submissions.

SETTING UP A CROSS-COMPILER

(for writing test-cases)

You cannot use ordinary gcc to compile ARM code (since it compiles for x86). You need to use a separate tool chain (or a gcc that is configure with target as ARM). You can use Linaro-gcc[3].

Download “gcc-linaro-aarch64-linux-gnu-4.9-2014.07_linux.tar.xz”. Extract it and you are good to go.

Note: 64-bit machines may require ia32-libs, lib32ncurses and lsb packages to be
installed (apt-get install lsb ia32-libs lib32ncurses5).

You can now compile using *aarch64-linux-gnu-gcc,* disassemble the binary using *aarch64-linux-gnu-objdump* and debug using *aarch64-linux-gnu-gdb*.

For those of you who want to run your ARM code, you could use QEMU [4]. QEMU is a hosted hypervisor but also has user-mode emulation for ARMv8. Download the latest version of QEMU and follow the steps:

```
tar -xvf qemu-2.1.0.tar.gz
cd qemu-2.1.0
mkdir build
cd build
../configure –target-list=aarch64-linux-user
make
```

```
./aarch64-linux-user/qemu-aarch64       <path-to-binary>
```

Note: The binary must be statically linked so use the   - -static flag while compiling.

DEBUGGING with GDB+QEMU

Generate debug symbols when you compile (-g compiler flag).
Add the -g 1234 option to qemu (./aarch64-linux-user/qemu-aarch64 -g 1234 <path-to-binary>). The -g option pauses it so that you can attach a debugger to it on port 1234. Go ahead and attach aarch64-linux-gnu-gdb to qemu's gdb stub on port 1234 by launching aarch64-linux-gnu-gdb and the typing the following commands at the gdb prompt:

```
gdb) file <path-to-binary>
gdb) target remote localhost:1234
gdb) list
```

And then you can proceed debugging (continue).

Instructions for setting up BitBucket:

Students are required to do regular check-ins in the *BitBucket* repository.
Else marks will be deducted.

Please follow the steps below to setup your repo:
Create an account on https://bitbucket.org/
Now create a private GIT repository armv8EPsim-repo
On the overview page you should be able to see commands to “Set up your local directory” to be a git repo. And using “Create your first file, commit, and push” you would be able to push a file online.
Below which there is link to add a team member. Please add your partner.
In settings -> access management please provide read access to following IDs:
csz148382@cse.iitd.ac.in, mcs142114@iitd.ac.in, mcs142123@iitd.ac.in, mcs142800@cse.iitd.ac.in.
Use git clone command to checkout the files from your repository.
Example in my case, user name 'siddhulokesh'

```
$ git clone https://siddhulokesh@bitbucket.org/siddhulokesh/armv8EPsim-repo.git
```

Please suitably modify the command as per your user name.

SUBMISSION DETAILS

Please ZIP all source files and then submit.
ZIP Name : <roll1>_<roll2>_CSPA1.zip.
Please check the moodle link for submissions.

DEBUGGER

Your simulator must have a debugging mode.

Use GDB and get a feel for it. Your debugger must have similar experience too.

To help you start, we will be providing you a debugger[1] with following functionality:

Activated by passing the “--debug” option on the command line. When in debug mode the simulator should stop so that the user can set breakpoints.

The following commands are supported:

break <ip> : puts a breakpoint at address <ip>. Ex: “`break0x40ab0400`”

del <ip>: removes breakpoint at address <ip>Ex: “`del0x40ab0400`”

run : starts the program execution

s : executes 1 machine instruction and pauses again.

c: continues execution till the next breakpoint is hit (execution is paused again)

print <x/d> [<eg> : Prints the contents of the register “reg” in either hexadecimal (x) or decimal (d)

Ex: “print x r3” should print the value of register r3 in hexadecimal.

print <num><b/w/d> <x/d> <mem-add>: Prints “num” bytes(b – 8 bits) / word(w – 32 bits) / double word(d – 64 bits) in hexadecimal (x) or decimal (d) from start of memory address <mem-addr>.

Ex: “`print4wx0x40ab0400`” will print 4 words from the start of `0x40ab0400` in hexadecimal.

watch <reg>: Stop execution after the first instruction that changes the value of register <reg>

Add the following commands to the debugger:

PHASE 1:

- cycles: Should return the number of cycles passed till now from start.

- stalls: Should return the number of cycles stalled till now from start.

PHASE 2:

- Previously coded commands like break, del, run, s, etc. Should work with the new pipe-lined execution of instructions.

- pipe: Should print the instructions in all the five pipe-stages.

- energy: Should return the energy consumed passed from start till now.

- activity: Should return the values of the activity counters for Instruction Cache, RF(Integer and Floating Point), Execution Unit(Adder, Multipliers, Dividers, Shifters of both Integer and Floating Point type), Data Cache.

- nc: executes next cycle and halts. More details will follow.

PIPE-LINING

A five stage single issue in-order pipeline has the following five stages:

Fetch from Instruction Cache

Decode instruction and RF read

Arithmetic/Logical Computation and Conditional Branch Evaluation

Access to Data Cache

Write-back to RF

<u>In-order Pipeline</u> : Instructions are not allowed to move ahead of previous instructions.

Computer Architects use various design techniques to keep all stages working in parallel however, the pipeline may stall due to 3 reasons:

Structural Hazard : Functional unit unavailable

Data Hazard : Operand unavailable

Branch Hazard : Taken branches cause fetching from new location. Hence instructions fetched after the branch are not useful and must be flushed. Also, restart fetching from new branch target address.

In your simulator there are various instructions which can stall the pipeline.

Figure them out. For branches, a hint is given below.

<u>Hint:</u>

A trace of a program is the sequential list of instructions executed.

In our case, the parser output is the trace. So for taken branches, you must add stall cycles.

A quick scan of Berkley slides[5] can be done. For more details, please looks at Anshul Sir's lectures[6] (Lec Num 24-27).

Using the above information you should be able to calculate the stall cycles, execution time and timing simulator would be ready. Now we have to modify the simulator to calculate energy. This is usually done using performance(activity) counters for each functional unit. These

counters keep track of the number of accesses of a functional unit in a given amount of time.
We will provide you the capacitance for each FU, you need to figure out the energy consumption. More details regarding DVFS and energy optimization will be given to you before Phase 3 starts.

Your simulator for 5 stage pipeline must support(at the end of assignment 1):

Forwarding and Hazard detection logic

Execution time and stall cycle calculation

Performance(activity) counter for each functional unit and energy calculation

DVFS based energy optimization scheme and EDP calculator

INSTRUCTION LIST

Already Implemented:

ADD, ADDS, ADR, ADRP, ASR, AND, B.{cond}, B, BR, BL, BLR, CBNZ, CBZ, CMP, LDP, LDR, LDRSW, LSL, LSR, NOP, MOV, RET, STP, STR, SUB, SUBS

To be implemented:

Phase 1 -

Conditional
CSET, CSINC, CSNEG
ALU
ADC, ANDS, BIC, BICS, CCMN,CINV,CLS, CLZ, CMN, CNEG
Load
LDRB, LDRH , LDRSB

Data Transfer
LDP, STP
Shift and rotate operations
ASR, LSL, LSR, and ROR
Unsigned Multiply Add Divide
SDIV,UMULL, UDIV

Phase 2 -

Mov
MOVK, MOVN, MOVZ,FMOV

Floating Point I
FADD, FSUB

Floating Point II
FMAX, FMIN

ENERGY MODELING

Following are the energy, power and delay modelling equations:

$P = C_L V_{DD}^2 \alpha f + t_{sc} V_{DD} I_{peak} \alpha f + V_{DD} I_{leak}$

P = Dynamic + Short Circuit + Leakage Power

We will neglect the short circuit power for now. So,

$P = C_L V_{DD}^2 \alpha f + V_{DD} I_{leak}$

P = Dynamic + Leakage Power

$E = CL\,VDD^2\,\alpha \;+\; VDDIleaktexecution$

E = Dynamic + Leakage Energy

$T = k/V$

So, we have given you one voltage, frequency and using the above modelling equations you need to

Estimate the voltage, power, energy, energy-delay product(EDP) for the frequencies below.

Draw a graph of EDP vs Voltage per functional unit.

Draw a graph of EDP vs Voltage for the processor.

Frequencies : 1000 MHz , 1500 MHz , 1600 MHz , 1800 MHz , 2100 MHz , 2200 MHz
Benchmark : Out of the five graded benchmarks[8] . You have to show energy modelling for one of the following(as per the completeness of your implementation):
bench6.s : If you are doing Float II instructions

bench5.s : If you are doing Float I instructions

bench4.s : If you are not able to do Float I or II instructions

CHECKLIST
High complexity/Time taking tasks are marked in Red.

PHASE 1 : SIMPLE INSTRUCTIONS

Register on Piazza to group “COP701: Software Systems Laboratory” at IIT Delhi in Summer 2015. https://piazza.com/

iit_delhi/summer2015/cop701
(AccessCode: cop701)

Form groups of 2.
And Fill the form: https://docs.google.com/spreadsheets/d/1BMygfHnxBTep5AMPJur8yAfeOxIghnqnkiHJ8rEhvKs/edit?usp=sharing

Learn Git

Learn Python

Setup BitBucket Account(Regular commits are required during all phases)

Start reading the stub[1] given by us

Setup the cross-compiler

Setup the debugger

Read the ARM manual

Start with implementing simple instructions

Understand and implement five stage pipeline with forwarding (assume, single cycle latency for each stage but model Mem-to-Reg data hazards)

Update debugger to show execution cycles, stall cycles

PHASE 2 : COMPLEX INSTRUCTIONS AND TIMING

Parse the config.xml[7]
Model multi-cycle operations.
Add Floating Point I and Floating Point II instructions to the simulator
Check stall modelling
Update simulator/debugger with performance(activity) counters
Update debugger handle phase 2 specs

PHASE 3 : ENERGY ESTIMATION AND EDP OPTIMIZATION

Write two test cases which check various instructions.

Do energy modelling at voltage stated in config.xml[7] and start modelling for other frequencies.

Make report.

More details will follow.

GRADING SCHEME

Activity	Marks
Regular check-ins to Bit-Bucket(along with relevant comments)	5
Simulator : Implementing Five Stage Pipeline	15
Debugger : Updating previous commands(break,del,run,s,c,print,watch)	10
Debugger : Implementing new commands(cycles,stalls,pipe, energy,activity,nc)	10
Four Graded Test Cases[8] given to you before submission	10
Develop two test cases of your own and verify simulator behaviour	8
Five Test cases(post submission)	22
Energy Modelling and Voltage Scaling Graphs	14
Max. Five page report having 2 lines about each source code file used in the assignment(previously + new) and at-least 3 known limitations of the assignment. Add results for your test cases in the report. Also energy, EDP graphs must be included for various frequencies.	5
Total	100

Bonus Marks: Those group which implement both Floating Point I and Floating Point II instructions
10

Bonus Marks: At least One Piazza Question/group
2

Bonus Marks: Finding and correcting bug in the previous code
Minor Bug : 1 + 2, Major Bug : 2 + 3, Hang : 2 + 5
Decision of minor, major, hang resides with the TA.
These bonus marks are per bug basis.

3/5/7

In phase 3 we will give you a set of 5 benchmarks/test cases. Hand calculate the execution/stall cycles and compare then with results from the simulator.
Please check for correctness of following: the register file & memory state, stall cycles, execution cycles and energy consumption.
Following is the composition of the benchmarks:

S. No.	Benchmark tests for
1	Previously implemented instructions
2	Phase 1 or 2 instructions(excl. Float I & II) [No data hazard will be there in the benchmark]
3	Phase 1 or 2 instructions(excl. Float I & II)
4	Phase 1 or 2 instructions(excl. Float I & II)
5	Phase 1 or 2 instructions(incl. Float I, excl. Float II)
6	Phase 1 or 2 instructions(incl. Float II, excl. Float I)

Benchmarks 1-4 are mandatory.
There is an option between 5 and 6, pick any one. If you do both, bonus marks will be awarded.
A similar set of benchmarks will be run post submission.
During the demo the TA will check the assignment and award marks.
The decision of the TA is final and binding(regarding the marks).

REFERENCES

ARM V8 Simulator, https://github.com/harrysethi/ARMV8_Simulator
ArmV8Arch,
http://infocenter.arm.com/help/index.jsp?topic=/

com.arm.doc.set.architecture/index.html (registration required to download the PDF)
http://releases.linaro.org/latest/components/toolchain/binaries/
http://qemu.org
https://inst.eecs.berkeley.edu/~cs152/sp09/lectures/L04-Pipelining.pdf
https://inst.eecs.berkeley.edu/~cs152/sp09/lectures/L05-PipeliningII.pdf
http://nptel.ac.in/video.php?subjectId=106102062
config.xml, https://drive.google.com/open?id=0B45IRMA1EBkSTXVLVEpNejQ2c1E
Graded Benchmarks,
https://drive.google.com/file/d/0B45IRMA1EBkSLXRGNlRsM1JJa28/view?usp=sharing
Git book: http://git-scm.com/book

ELF Format: http://www.cse.iitd.ac.in/~sbansal/os/ref/elf.pdf

Makefile Tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

For anything not mentioned - Google and Stackoverflow are your best friends