Design of Embedded DSP Processors

Unit 7: Programming toolchain
Toolchain introduction

There are two kinds of tools

1. The ASIP design tool for HW designers
   - Frontend tool: Profiler and architecture optimizer
   - Backend tool: Processor (ASIP) synthesizer
   - Pipeline accurate simulator: For HW debugging

2. The assembly coding tool for programmers
   - C-compiler, Assembler, Linker, (Loader)
   - Instruction set simulator: for ASM programmers
   - ASM coding Debugger: for firmware designers
Job loads to develop an ASIP

- Processor HW architecture
- Toolchain and kernel lib
- SW Ecological env
Understand Applications
Function and Architecture Specification
ASM Instruction set Specification
Cost and Performance Estimation
ASIP Designer Synthesizer
Benchmark Synthesized ASIP
Functional Verification
Cost, Performance and Power Estimation
VLSI Implementation
Silicon Cost and Power Release

Source code
Lexical analysis, parsing
CFG extraction
Compiler
CFG
Annotation
Static profiling
Dynamic profiling
Stimuli
Critical path
ASIP design document

Good to know, not for exam
Lexical analysis
Parsing
Parse tree optimization

Parse tree

CFG extraction

CFG

Basic block (bb) cost estimation

Annotated parse tree

bb cost table

Code generation

Linking

Compiler front-end

Compiler back-end

Using GCC

Probe lib

Available profiler: Relief for application analysis

GCC: GNU Compiler Collection
CFG: control flow graph
GEM: GNU C Compiler Extensions Framework
Synopsys ASIP Designer

Processor Model

- primitive header file
- nML file
- PDG file
- compiler header file

Compiler
Assembler
Linker
Simulator
RTL
ASM coding tools
ASIP User's View on tools

Behavior (C code) modeling (1)(2)

Function allocation to HW (1)(2)(3)

Behavior code adapt to HW (1)(2)(3)

Build assembly program library

Compile C code to assembly (4)

Assembly programming (5)(6)(7)

Link objective codes and allocate codes to program memory (6)

Generate finally binary code and simulation using the binary code (7)(8)

Verify codes, benchmarking, profiling, and release the design (5)(6)(7)(8)

(a) The method translating source code to qualified assembly code

(b) Toolchain
The focus in unit 7 today

• Briefly introduce compiler, because there is another compiler course

• Focus on Simulator
  – Cycle accurate (microarchitecture) simulator
    • For HW development and verifications
  – Instruction set (behavior) simulator ISS
    • For firmware implementation and debugging
    • Trace driven simulator: fast sim, no-exec high MI/S
    • Execution driven simulator: easy design for ASIP
What is a compiler

• A compiler is a program that reads a program written in one language (usually a high-level language) and translates it into an equivalent program in another language (usually a low-level language)

• For an ASIP programmer, the source language is C and the target language is ASM
Compiler structure

Source program

String of characters

Lexical analyzer

String of tokens

Parser

Parse tree

Semantic analyzer

Parse tree /AST

Symbol-table manager

Error handler

Analyzer

Synthesizer

Intermediate code generator

Intermediate representation

Code optimizer

Intermediate representation

Code generator

Intermediate representation

Relocatable assembly code

Target code optimizer

Target program

Target code optimizer

Relocatable assembly code
Compiler structure

• Compilation consists of two phases:
  analysis and synthesis.
  – Analyzer: It breaks the source program into consecutive pieces and conducts lexical, syntactical, and semantic analysis.
  – Synthesizer: creates intermediate representation (IR) versus source program. IR Code is then optimized on IR level. Finally, ASM codes (glibc call) are selected based on the optimized IR code.
Frontend analyzer

- Lexical analysis and parsing C program, early code filtering, and generate abstract syntax tree /control flow graph.)
Code optimization

• Most compiler researches are optimizations
  – So many papers can be found, such as……
  – Data Flow Optimizations: identify and handling independent operations
  – Common sub-expression elimination: eliminate more expressions compute the same value
  – Loop Optimizations: minimize the inner-most loop runtime, M size, or memory access cost
Code generation

• The final step in a compiler is target code generation and optimization
  – Instruction Selection: match IR operations to target template (in the template table)
  – Register Allocation: Allocate variables and generate load store instructions
  – Instruction Scheduling: improve instruction-level parallelism via pipeline
Assembler as a program

• Assembler is a computer program
  – Translates assembly instruction mnemonics, memory locations, and other entities into binary machine code.
  – The input of an assembler is assembly code.
  – The output file is an object file, containing relocatable binary codes and bookkeeping information (symbol table for the linker).
Assembler design/construction

• The translation consists of two steps.
  1. Find memory locations with labels: by calculating addresses of all symbolic address names.
  2. Translate each assembly statement, mnemonic, register, configuration switch, and label into binary code
• Finally, the assembler produces an output file (object file), containing the machine instructions, data, and bookkeeping information.
Load an assembly source file (*.ASM)

Load initial symbol table (reserved mnemonics)

1st pass

Read token from source (merge lexical and syntax analyses)

Token in symbol table?

- y: Redefinition of token

- n: Add token to symbol table

Add token to symbol table

Add token to ASM expression

ASM expression complete

- y: Concatenate code to internal code list

- n: End of source?

2nd pass

Get code from internal code list

Does code need identification / label patching?

- y: Is identification / label defined in symbol table?

- n: End of Code list

- y: Patch code in code list with value of identification / label

3rd pass

Find memory locations with labels: by calculating addresses of all symbolic address names

Generate machine code and bookkeeping symbol table as the output from internal code list
Linker as a program

• A linker is a program, it takes object modules with linking tables and library / subroutine codes as inputs, merges them, generates final (code and data) addresses, executable binary machine codes.

• By separating assembler and linker, code can be re-locatable
Linker function list

• A linker performs the following tasks:
  – Searches the program libraries to find library routines used by the program
  – Determines the memory locations of codes from all modules and relocates instructions by adjusting absolute references
  – Resolves references among files (match all labels to physical addresses and calculate all target addresses of jumps and calls)
Linker, functional view

Codes

relocation record

Main:
  ...
  Jump to ???
  ...
  Jump to ???

Call.sub1
Call.blockLoad

Sub1:
  ...

Lib:
  ...
  BlockLoad

The linker

//executable
Main: ...
  Jump BlockLoad
  ...
  Jump Sub1
Lib:
  ...
Sub1:
  ...

Codes relocation record
Summary for tools mentioned

• Compiler, assembler, and linker are tools based on lexical, syntactical, and semantic analysis.
• Assembler translates symbolic instructions into binary and creates relocation bits
• Linker creates executable from several files produced by assembler.
• There are many available tool platforms, you carefully select one for your long term developments (GNU, LLVM), opencircuitdesign.com……
• Much from free & open source SW (FOSS)
Simulator

The focus of the lecture
Simulator definition

• Simulator in general: A program designed for simulation
  – Modeling a real-life situation on a computer to study and to see how the system works.

• A processor (ASIP) simulator
  – It is a program modeling a dedicated instruction-set or processor HW to debug HW / FW designs, predict performance metrics on a given input
A processor simulator

Core simulator

program memory

data memory

interrupt
DMA

timer
I/O

Processor simulator

Debugger

code

data
Simulator Inputs

1. Assembly program in binary code format, loaded into the program memory.

2. Data include inputs, coefficients, and program parameters, loaded to the data memories.

3. IP configurations for a design. For example: DM0=32kx16b, DM1=8kx16b, PM=4kx32b.

4. Configurations and assignments of peripheral devices in a design.

5. Input from debugger, e.g., run mode (stepwise, run N cycles, stop somewhere).
Simulator Outputs

1. Data stored in all addressable registers, and memories when the simulation is stopped.
2. Current status of the processor (PC, flags, processor status, control registers, stack pointer, stack data) and status of the finite state machines.
3. Error report if there are errors.
Simulator classification

Instruction Set Simulation

Trace driven

Execution driven

Interprite

Binary translation

Mixed

For most COTS

ASIP designers usually design execution driven interpriter simulator
Instruction-set simulator, ISS

• For software developers!
• With a cycle counter, it can get roughly cycle cost (if there is no HW stall)
• Bit accurate (including guard / saturation / round / carry-in / scaling)
• Emulate functions in data / data access paths
• Not yet pipeline accurate in control path
How to design an ISS

1. Run BIN code instead of ASM code (real processor)
2. Cycle counter (accumulate cycle cost), memories & RF as global variables exposed to debugger / GUI
3. Fetcher: A PC FSM emulator to get next-instruction (and a function call based on function variables)
4. Decoder and Executer: Decode and decide which instruction \((\text{function})\) to call. Priority: a) debugging, b) exception, c) interruption, d) next instruction.
5. Instruction emulation: Coding each instruction as a \textit{function}, No emulation of HW sharing.
If it is not in debug mode, set the start point to PC, set run mode, START

- Fetch an instruction, extract type code of the instruction

  - Branch/call
    - Y
  - Register ↔ memory
    - Y
  - CISC *
    - Simple loop
      - Y
    - Other CISC

  - ALU REG subroutine
    - Data dependent
      - Y

  - Register to Register OP
    - PC++
  
- PC <= exception/interrupt service entry

  - Condition
    - Y
  - Target address CC=3
    - Stack OP for Branch Call and Return
      - Overflow
        - Y
      - Jump OP
        - NPC
      - PC++

- CISC subroutine
  - Memory Addressing Data access
    - Overflow
      - Y
    - CISC OP
      - Loop
        - Finished
          - PC++

  - Load store OP
    - PC++

- Load store subroutine
  - Addresses
    - Overflow or data dependent
      - Y

- Data dependent
  - PC <= exception/interrupt service entry

- Store status, release destination address, and count clock cycles CC=3, or 2, or 1 (default)

- Debugging mode
  - N
  - To debug mode

* CISC modeling can be different
ISS implementation issues

• The main
  – The main platform, the global variable design for function call and for debugger I/O
  – Top level modules: fetcher, decoder, executer, memory & register manager

• Functions
  – Each instruction is emulated by a function (call) and being called while running it.
1. Implement fetcher, decoder, execution unit, and M-manager

- Fetcher: Behavior of a PC FSM
- Decoder: Decoding binary instruction, and prepare for parameters of an instruction. It might be the performance critical path
- Executor: Call a function
- M-manager: Data memory for different types, debugging and tracing interface.
2. Implement an instruction

// Show in ASM code, it is actually BIN)
return_type function_name( parameter list )
{ body of the function }

// E.g. Define a function in simulator
function ADD(DEST OPA OPB)
{ Int DEST, OPA, OPB
  DEST = OPA + OPB}
endfunction

// The function used in simulator
ADD (R2 R0 R1) // R2=R0+R1, R0,1,2 are in RF
3. Implement the cycle counter

- Cycle_counter is a global variable
- Count +1 to run a normal instruction
- Count +n for special instructions, such as a taken jump
- A profiler (cycle cost, extra non-function cycle cost, each instruction appearance) might in the counter
4. Implement Debugger interface

• Debugger is an even higher level program over the ISS, it is an I/F to programmers

• It will be explained later in this unit

• Necessary to mention “debugger” here
  – to sufficiently expose parameters during ISS design, “debugger” can thus access / control
  – to load data and program codes, configure for a run, observation, resume.
Problems to avoid

• Functions that not designed in HW (yet can be executed in C) shall not be executed by ISS, For example (check illegal functions using ABT):
  – Move data between special registers
  – Move data between special registers and DM

• Bit accurate emulation and their orders:
  – Computing based on 40b long data in MAC
  – Guard, scaling, rounding, saturation, truncation
Host adaptation for performance

• If the memory in designed machine is larger than the host main memory, the host cache mismatch will happen frequently, how to avoid it.

• Maximuly adapt simulated instructions to host instructions (binary translation, low portability) and minimize interpritated instructions.

• Many ways to minimize ”instruction decoding time” in host (static pre-decoding table etc.)
Microarchitecture Simulator

- HW-SIM is for hardware design / debugging
- Cycle and (function in each) pipeline accurate
- Bit and order accurate (including guard / scaling / round / saturation / truncation / carry-in/out)
- Emulate all functions in datapath, data access path (emulate HW sharing to avoid structure hazard)
- Pipeline accurate for controls (jump, NOP, loop)
Microarchitecture Simulator

• Similar to ISS, the difference is that the simulator is “hardware” running functions in each pipeline step.
• Clock and cycle_counter are global variables
• In each “hardware” pipeline step, there are IN and OUT of the pipeline.
• When clock changes, The IN in the next pipeline get data from OUT of the previous pipeline
How to write HW-SIM platform

• Function of each instruction shall find available HW in each pipeline step supporting for the function:
  – There will be a structural bug if there is no related hardware. Each hardware module can be used only once in a pipeline.
  – The function (an instruction) is executed in the way to run the first pipeline function in the first pipeline, and pass all other functions to the next pipeline,
  – In each pipeline, functions for my pipeline is executed and the rest is passed to the next pipeline.

• Emulate for hardware sharing? Up to you. If you need to check structural hazard.
Pipeline accurate implementation

• For example, on top level:
  – One global signal: `machine_clock // a counter`
  – `machine_clock <= machine_clock + 1; // clocking`

• For example, in each module:
  – After all combinational functions are executed,
    If `machine_clock != machine_clock_old`
    • `machine_clock_old: local variable in each module`
    • All flip flop `Q <= all flip flop D`
    • `machine_clock_old <= machine_clock`. 
How to write HW-SIM functions

• For time stationary processors, instruction functions shall be distributed to each pipeline step

• To support for hardware debugging
  – All hidden functions (instruction fetch, instruction decoding, data fetch) shall be explicitly pipelined
  – All operations of an instruction (including hidden functions) are partitioned into each pipeline step

• This is only an example. You can design it in your way
Hidden functions as examples

All micro-operations in an assembly instruction

Explicit micro-operations specified in assembly manual:

Explicit micro-operations specified in assembly code and binary machine code:

Data memory addressing
Operands
Destination
Operation
Explicit specifiers
Target addressing

Implicit micro-operations not specified in assembly code:
For example flag ops and PC<=PC+1

Implicit micro-operations:
for example bus transactions, and instruction decoding

Data memory addressing
Operands
Destination
Operation
Explicit specifiers
Target addressing
Core simulator

- It can be ISS or HW-SIM
- It is the “naked” simulator without I/O
- Sizes of the basic I memory and D memory are the largest (limits of the instruction set)
- Simulator is part of the core IP deliverable
- It will be integrated in system SoC
Processor (SoC) simulator

• Configured for a project and defined by an IP user
• With dedicated I/O peripherals and custom memory size
• Addressing overflow shall be checked for the IP configuration.
Normal mode and debug mode

• The simulator keeps checking execution errors, when there is an error, the simulator goes to the debug mode

• Errors include (at least)
  – Program memory address (PC) overflow
  – Data memory address overflow
  – Data dependent, overflow, underflow etc.
  – Register file size and addressing problem
  – Stack size, overflow and underflow problem
Finally: Using available platforms

• Available platforms for low speed ISS, e.g.:
  – Shade: Trace-driven, instruction set simulator (may not for us)
  – SimpleScalar: execution-driven, cycle-accurate simulator
  – SPIM: execution-driven, instruction set simulator
  – SMTSIM: execution-driven, cycle-accurate simulator
  – MikroSim: define instruction set on microcode level
  – OVPsim: >100 instruction sets, can define own instruction sets
  – GEM5: For in order / out of order arch, from U-Michigan

• To build an ISS using Synopsys ASIP designer
Debugger
GUI and IDE
Debugger

• A debugger is a program to test and debug assembly programs.
  – Load assembly code and data
  – Programmer interface
  – Support simulator execution and debugging

• Get an available debugger for free (Eclipse from IBM, GDB from GNU……)
Debugger

• A debugger supports running modes:
  – step-by-step mode (running single-stepping),
  – N-instruction mode (running N instructions),
  – Breakpoints mode (pausing at breakpoints).

• A debugger supports observations of
  – All data in RF and in M when program stopped

• Keep values and continue execution
Review the discussion today

• A processor cannot be used without enough supports by FW programming tools.

• Compiler, assembler, linker, simulator, and debugger were introduced.

• The introduction was prepared for toolchain users, insufficient for toolchain developers.
## Summarize what/how to learn

### Skills

<table>
<thead>
<tr>
<th></th>
<th>System understanding</th>
<th>FW coding</th>
<th>Integration verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly coding tools</td>
<td>lexical, syntactical, and semantic analysis</td>
<td>debug</td>
<td>How to used the tool chain for verification</td>
</tr>
<tr>
<td></td>
<td>Pipeline accurate simulator ISS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Firmware plan &amp; design</td>
<td>The relation between firmware design flow (for main and for kernels) and the firmware design tools</td>
<td>Firmware design flow</td>
<td>------</td>
</tr>
</tbody>
</table>
Self reading after the lecture

• How to design a simulator for firmware developers
• How to design a microarchitecture sim for hardware debugging
• Read through Chapter 8
• Much more to read for tool developers
Exciting time now!

Let us discuss

• Whatever you want to discuss and related to HW
• You will have the chance after each lecture (Fö), do take the chance!
• Prepare your Qs for the next time
Welcome to ask any questions you want to
• I can answer
• Or discuss together
• I want to know what you want

Questions are guaranteed in life; Answers aren't.

Dake Liu, Room 556 corridor B, Hus-B, phone 281256, dake.liu@liu.se