09 – The Toolchain

Oscar Gustafsson

Todays lecture

- DSP toolchain (assembler, compiler, linker, simulator, debugger)

Toolchain

- Assembler
- Linker
- Compiler
- **Simulator**
- Debugger

Assembler

- Simple explanation
    - Read assembler source code
    - Translate to machine code
    - Sounds easy?
- Any programmer can write a **simple** assembler

Assembler - more complete explanation

- Repeat: Read one line of source code
  - Remove comments
  - Is this an assembler directive? If so, handle it
  - Is there a label on the line?
    - Add to label database with the current address
  - Is a label used anywhere on the line?
    - Is the address of the label unknown?
    - Add address to unresolved label database
  - Save (possibly incomplete) machine code of line to memory
- Once the entire file has been read: Fix all unresolved label references and write out final machine code to an output file

Assembler - tricky things

- It is very convenient to be able to use expressions like this:

```
.equ HSIZE 16
.equ PITCH 160
load r0,label+(5*HSIZE+PITCH*10)/4
```

- This requires the assembler to parse complex expressions (more on parsing later)

Assembler - More tricky things

- On many architectures there are two kind of jump instructions
  - A single word instruction: `PC = PC + offset`
  - A double word instruction: `PC = absolute_addr`
- Before you have resolved all labels you do not know if the offset is small enough to fit a single word jump

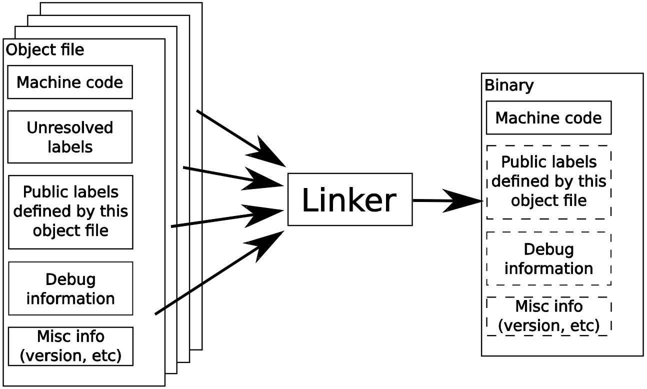
Assembler - More Tricky things

- When changing a single word instruction to double word you will have to redo many other label calculations as well!
- This can also be true for other kind of instructions
  - For example, load/store with absolute addressing:
    - First 1024 words can be accessed using 1 insn word
    - The remaining memory requires 2 insn words

Linker

- When implementing larger applications the application is usually divided into several parts or libraries
  - All labels will not be resolved by the assembler when assembling a single source file
  - Surprisingly non-trivial for the general case (dynamic libraries, etc)

Linker functionality



”I don’t want to write a linker”

- Invoke a preprocessor like CPP from your assembler
- This is a decent substitute for a linker:

```
/* Start of main assembler file */
#include "iolibrary.asm"
#include "fft.asm"
#include "huffman.asm"
// Bonus 1: Comments can be handled (removed) by CPP
// Bonus 2: You do not need to implement EQU yourself:
#define BLOCK 16

main:
  call iosetup
  call fftsetup
  ...
```

”I don’t want to write a linker”

- Not perfect, you still have to handle:
  - Labels
  - (Expressions)
- Cannot handle pre-assembled libraries
- It is enough to get your software developers started before a linker is available

Compiler

- For most users a compiler works like this:
    - Read source code
    - Compilation process *(MAGIC ???)*
    - Output machine code

Compiler

- Slightly more advanced view
  - Read source code
  - Compilation process (MAGIC ??)
  - Output assembler
  - Run assembler
  - Run linker

Compiler crash course

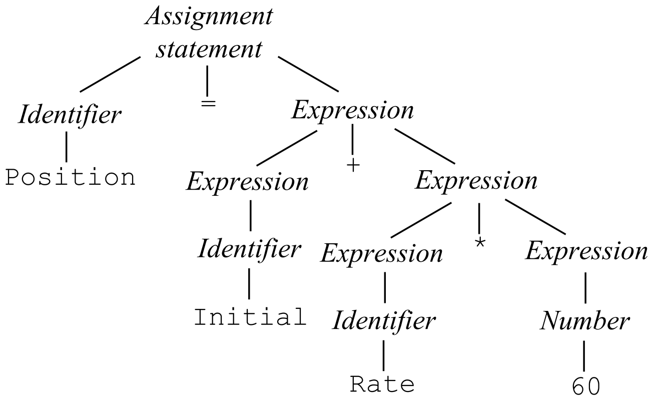
- Frontend
  - Read source code and tokenize it (lexical analysis)
  - Parse source code and generate abstract syntax tree
- (Middleend?)
  - Optimize (MAGIC?)
  - Code generation
- Backend
  - Output assembler
- Not a part of the compiler
  - Run assembler
  - Run linker

Tokenizer (lexical analysis)

- All lines in the source code are divided into tokens of different types

```
position = initial + rate * 60; // Comments are ignored

    TOKEN_SYMBOL("position")
    TOKEN_ASSIGN
    TOKEN_SYMBOL("initial")
    TOKEN_PLUS
    TOKEN_SYMBOL("rate")
    TOKEN_ASTERISK
    TOKEN_VALUE(60)
    TOKEN_SEMICOLON
```

Parse tree for position = initial + rate * 60



Lexical analyzer/Parser summary

- Fairly tricky to write by hand
    - Depends on the source language (e.g. Lisp vs C++...)
- Luckily there are tools to do this for us:
    - Flex - Generate a lexical analyzer
    - Bison - Generates a parser
    - Knowing how to use these (or similar) tools is an important skill for every programmer!
    - You will learn to use these tools in the compiler construction class at IDA

Lexical analyzer/Parser summary

- You will learn to use these tools in the compiler construction class at IDA
  - **Read this course!**
    - **(Or learn it by yourself)**

Lexical analyzer/parser

- Any programmer may need to parse files at some point
- Aside the obvious (parsing source code in various languages):
  - Parsing configuration files
  - Parsing log files
  - Making a command line interpreter
  - etc…
- Knowing how to use lex/yacc (or similar tools suitable to the programming language you are using) will save you a lot of time!

Optimization phase

- You do not need to worry about this phase
- Interesting, but we'll consider it a black box with some sort of magic inside in this course

Code generation

- Take optimized representation of program and output assembler program
- If we have a production compiler like GCC or LLVM this is the only part we need to modify to port it to our processor

Backend porting is not magic (Example from GCC)

- Easy issues
  - Datatype sizes (sizeof int, short, char, etc)
  - Big/little endian
  - Number of registers, different register classes

GCC backend porting is not magic

- Not so difficult
    - Instruction patterns for basic operations like move, add, sub, multiply, etc

GCC backend porting is not magic

- Fairly tricky
  - Stack frame format and calling convention
  - How memory addressing works in your CPU

GCC backend porting issues

- At first nothing works at all
  - This is the difficult phase, there are lots of settings you need to tweak for your processor
- Once GCC generates code it is easy to incrementally improve your backend

GCC backend - Timeframe

- Count on 3 months to get a working non-optimized backend up and running
- Count on 3 more months to get a decent optimizing backend for a RISC-like processor
- Do not count on being able to create a backend that can output specialized ASIP instructions

Simulator vs Emulator

- Historically:
  - Emulator - hardware is involved to emulate another system
  - Simulator - done in software
- Today - more confusing:
  - In mainstream usage, an emulator is more or less the same as a simulator (e.g., NES emulator)
  - In the SoC society, emulation is usually associated with huge FPGAs that emulate an ASIC before fabrication

Behavioral Simulator

- Read machine code from file
- Repeat
    - Read instruction
    - Interpret instruction
    - Execute instruction

Behavioral simulator

- Requirements
    - Bit accurate
    - As cycle true as possible
    - Relatively fast
- Mostly used for software development

Remember this example?

Mnemonic	Encoding
`ADD rD,rS,rT`	`0000 ssss tttt dddd`
`SUB rD,rS,rT`	`0001 ssss tttt dddd`
`CMP rS,rT`	`0010 ssss tttt 0000`
`MUL rD,rS,rT`	`0011 ssss tttt dddd`
`JMP A`	`0100 0000 aaaa aaaa`
`JMP.EQ A`	`0101 0000 aaaa aaaa`
`JMP.NE A`	`0110 0000 aaaa aaaa`

Main loop

```
void simulate(void) {
  read_machine_code("inputfile.bin");
  initialize_registers();
  while(1) {
    execute_instruction();
  }
}
```

execute_instruction()

```
uint16_t insn=read_insn();
increment_pc();

int rs, rt, rd;
rs = (insn & 0x0f00) >> 8;
rt = (insn & 0x00f0) >> 4;
rd = (insn & 0x000f);

uint16_t opa = rf[rs];
uint16_t opb = rf[rt];
switch(insn & 0xf000) {
  case 0x0000:
    do_add(opa,opb,rd);
    break;
```

```
case 0x3000:
  do_mul(opa,opb,rd);
  break;
case 0x4000:
  check((insn & 0x0f00) == 0);
  do_jmp(insn & 0xff);
  break;
case 0x5000:
  check((insn & 0x0f00) == 0);
  do_jmp_eq(insn & 0xff);
  break;
case 0x6000:
  check((insn & 0x0f00) == 0);
```

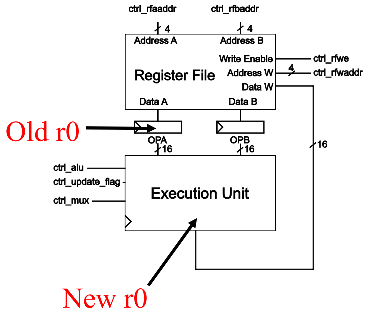
Execution

```
void do_add(uint16_t opa, uint16_t opb, int rd) {
  uint16_t result = opa + opb;
  rf[rd] = result;
  if(!result) {
    zflag = 1;
  } else {
    zflag = 0;
  }
}
```

Pipeline effects

- The simulator can handle code with data dependencies that the hardware cannot

```
add r0,r1,r2
add r4,r0,r3
```



Protecting the register file

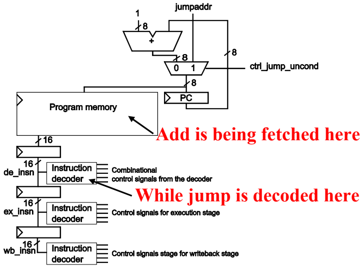
```
int rf_busy[16]; // Keeps track of whether a
                 // result is ready
void write_reg(int regno, uint16_t value, int delay) {
  rf[regno] = value;
  rf_busy[regno] = delay;
}
void every_cycle(void) {
  for (int i = 0; i < 16; i++) {
    if (rf_busy[i] > 0) {
      rf_busy[i]--; // If busy, decrement
    }               // busy counter until we
  }                 // are allowed to access
}                   // the register again.
```

Protecting the register file

```
// Read from register file while checking if the
// result has been written
int get_opa(uint16_t insn) {
  int rs = (insn & 0x0f00) >> 8;
  if (rf_busy[rs] > 0) {
    err("FAIL:␣Accessing␣a␣register␣before␣it␣is␣ready");
  }
  return rf[rs];
}
```

Pipeline effects for jumps

- Remember this example?
  - `jmp 0x59`
  - `add r5,r2,r3`
- Jumps have one or two delay slots in our simple example


jumpaddr
1
8
+
8
8
0 1
ctrl_jump_uncond
8
PC
Program memory
Add is being fetched here
16
16
de_insn
Instruction decoder
Combinational control signals from the decoder
While jump is decoded here
16
ex_insn
Instruction decoder
Control signals for execution stage
16
wb_insn
Instruction decoder
Control signals stage for writeback stage

Delay slot handling

```
// Delay slot handling
int delay_slot = 0;
uint8_t newpc = 0;
void increment_pc() {
  pc = pc + 1;
  if (delay_slot) {
    delay_slot--;
    if (!delay_slot) {
      pc = newpc;
    }
  }
}
```

```
// Conditional jump if equal
void do_jump_eq(uint8_t addr) {
  if (zflag) {
    delay_slot = 2;
    newpc = addr;
  }
}
```

Summary: Behavioral simulator

- It is possible to write a behavioral simulator that is bit true and cycle true without implementing a complete pipeline
    - (Although a behavioral instruction set simulator may be implemented without being completely cycle true)

Micro-architecture simulator

- This simulator must be:
  - Cycle true
  - Bit accurate
  - Pipeline accurate
- This simulator will be slower
  - Used for verification of RTL code

Micro-architecture simulator

- The simulator is partitioned like the processor pipeline

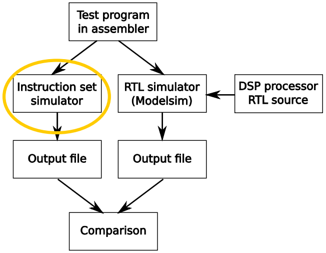
```
void run_one_clock(void) {
  pipeline_fetch_insn();
  pipeline_read_operands();
  pipeline_execute();
  pipeline_writeback();

  update_all_pipeline_registers();
}
```

Micro-architecture simulator

- Debug functions to readout the contents of various pipeline registers
    - Especially important when debugging large and complex processors (like out-of-order superscalar)

RTL simulation and the golden model

- When you are reasonable sure that no bugs are left in the ISS you can use this as the golden model that the RTL source code must conform to



Simulator features: Snapshots

- The ability to save the entire state of the simulator and reload the state at a later time

Simulator features: Profiling

- When running a program the simulator can increment a counter for each instruction every time it is executed
  - You can also profile other things like branch taken/not-taken probabilities, memory usage, estimated power usage, etc

Simulator features: Reversability

- Some simulators can step back in time
  - Very nice for debugging
    - Example: Run backwards to determine where a certain pointer was set to illegal value
- (Hard to implement efficiently without snapshots.)

Simulator feature: Tracing

- Create a log of important events such as I/O, memory read/writes, conditional branches, etc
- Create a VCD (value change dump) file for wave form viewer

C and ISS co-simulation

- Scenario: You are developing a JPEG encoder together with a few other engineers
  - Engineer 1: Writing assembly for Huffman encoding
  - Engineer 2 (you): Writing assembly for DCT
  - Engineer 3: Writing assembly for main program
- You are basing your encoder on some sort of reference code

C and ISS co-simulation

- Problem
  - You cannot easily test your assembly code unless engineer 1 and 3 are finished with their tasks
- Solution
  - Make sure your simulator can be loaded as a library from C
  - Use reference JPEG encoder for everything but the DCT

C and ISS co-simulation

```
#ifdef COSIMULATION
static int loaded = 0;
if (!loaded) {
  sim.assemble("dct.asm");
  loaded = 1;
}
for (int i = 0; i < 64; i++) {
  sim.write(input[i], sim.addrof("input") + i);
}
sim.setpc(sim.addrof("dct"));
sim.run();
for (i = 0; i < 64; i++) {
  output[i] = sim.read(sim.addrof("output") + i);
}
#else
  // Normal DCT goes here
#endif
```

C and ISS co-simulation

- Problem solved
    - All engineers can test their code without worrying about bugs in the others assembly implementations

RTL and ISS co-simulation

- It could be very interesting to be able to start development of an SoC system before the RTL code of the DSP processor is done
- **This is very important for larger systems**
  - Allows software development to start early!
  - Simplifies debugging of SoC hardware before processor is ready
- This can be done by (for example) allowing the ISS to be called from Verilog or VHDL
  - In Verilog this is called VPI or PLI

Handling custom instructions through plugins

- It is desirable to have a plugin interface to your assembler and instruction set simulator to allow for the development of custom instructions and/or accelerators
  - Especially if you do not want to expose your toolchain source code to your customers

Debugger

- Some requirements
  - Single step
  - Breakpoints based on program counters
  - View source code for current assembly instruction
  - Breakpoints based on data value on bus

Debugger for ISS

- Just add a few functions. Most of the functionality is already there.
- For more advanced debuggers you may want to follow already established ad-hoc standards for debugging
  - For example, gdb debug protocol

Debugger for real hardware

- Two possibilities
  - Implement debugging in software running on the DSP core
  - Implement debugging in hardware, software is not involved at all

Debugger for real hardware

- Breakpoints in software debugger
  - PC FSM modified to generate exception after one instruction when singlestep is activated
  - PC FSM modified to generate exception when reaching a certain (configurable) value
  - Comparator on data buses to generate exception when a configurable value is present

Debugger for real hardware

- Hardware debug unit
  - All debugging goes over a dedicated debug interface (typically JTAG)
  - When stalled the RF, data/program memories, PC, and even the pipeline registers can be read/written over the debug interface
  - Must still have support for hardware breakpoints, but no need to take an exception
  - May support advanced features such as tracing as well

Debugger for RTL code

- If you have support in the hardware for a debugger you can use the same support to debug programs executing in an RTL simulator

Time to start thinking about the exam?

- How to prepare:
  - Solve tutorial exercises
  - Solve the design challenge in the exercise book
    - Cross-check your solution proposal with a friend
  - (Study old exams…– See the tutorial exercises)

Questions about the exam?

- Feel free to visit me in my office
  - You might want to email me in advance if you want me to be more prepared for your answer
- You can of course also ask me or Erik

Oscar Gustafsson

www.liu.se

LINKÖPING UNIVERSITY