Computer Organization Final Project

Andrew Zonenberg

Overview

- Goals of project
- Compilation/cross-compilation workflow
- ELF executable format
- Simulator structure

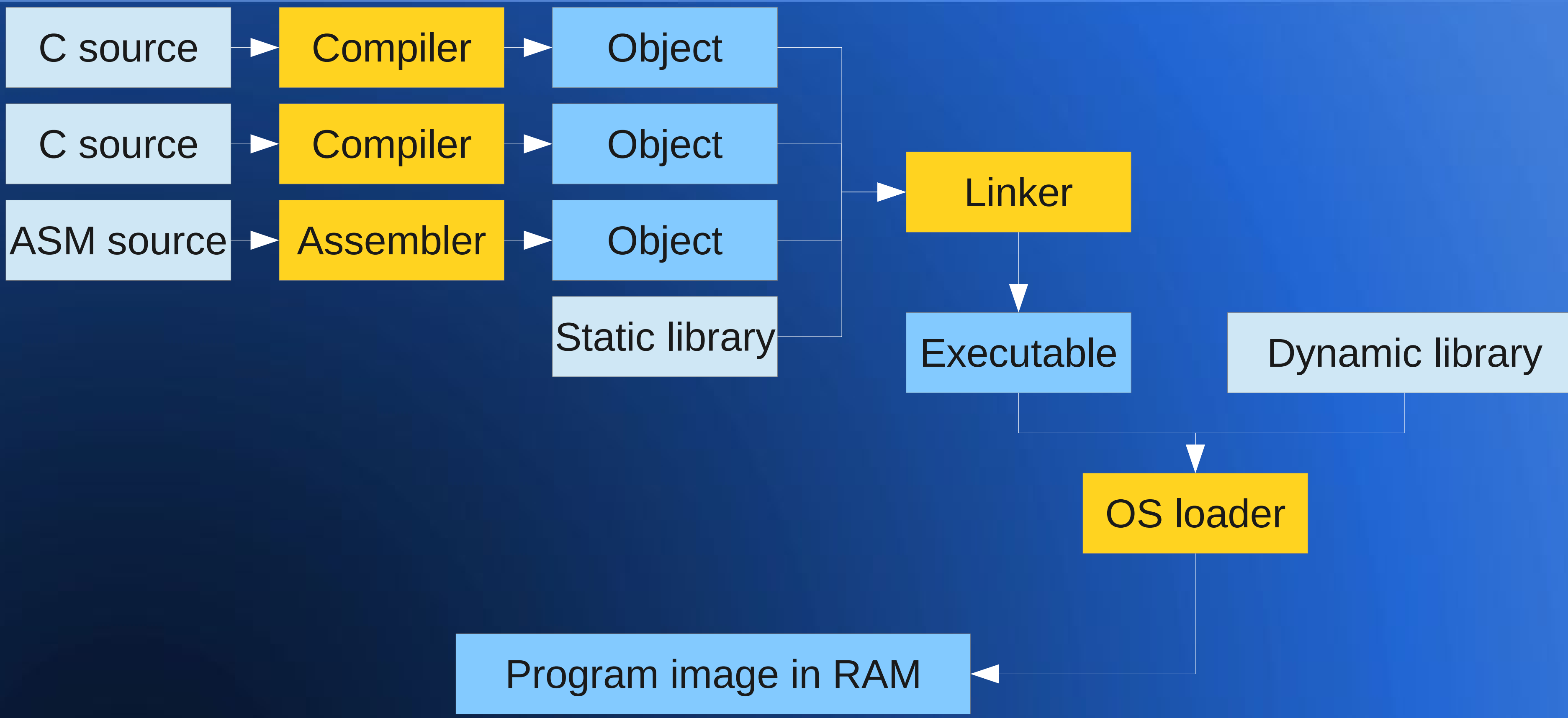
Project goals

- MIPS CPU simulator - like SPIM but simpler
  - No single-step mode
  - No register/memory views, just run syscalls
  - No built-in assembler, use gcc instead

Compilation process

- Preprocessor
  - 1 C source -> 1 preprocessed C source
- Compiler
  - 1 preprocessed C source -> 1 asm source
- Assembler
  - 1 asm source -> 1 object
- Linker
  - N object -> 1 executable

Compilation process



Preprocessor

- Insert contents of #include files
- Remove comments
- Expand macros

Compiler

- Generate AST from preprocessed source
- Generate assembly code for each function
- Optimize generated code

Compiler optimizations

- Simple, naive compilation emits one or more asm instructions per line of C without considering surrounding code
- Lots of redundant loads/stores
- Code may be duplicated
- Optimization tries to remove this. Generated code is smaller and faster, but may be harder to read

C source

```
int foo(int a, int b)
{
int c = a+b;
return c;
}
```

Unoptimized assembly

```
int foo(int a, int b)
{
   0: 27bdffe8    addiu   sp,sp,-24
   4: afbe0014    sw      s8,20(sp)
   8: 03a0f021    move    s8,sp
   c: afc40018    sw      a0,24(s8)
  10: afc5001c    sw      a1,28(s8)
int c = a+b;
  14: 8fc30018    lw      v1,24(s8)
  18: 8fc2001c    lw      v0,28(s8)
  1c: 00200825    move    at,at
  20: 00621021    addu    v0,v1,v0
  24: afc20008    sw      v0,8(s8)
return c;
  28: 8fc20008    lw      v0,8(s8)
}
  2c: 03c0e821    move    sp,s8
  30: 8fbe0014    lw      s8,20(sp)
  34: 27bd0018    addiu   sp,sp,24
  38: 03e00008    jr      ra
  3c: 00200825    move    at,at
```

Optimized assembly

```
0:  00a41021    addu    v0,a1,a0
4:  03e00008    jr  ra
8:  00200825    move    at,at
c:  00200825    move    at,at
```

Assembly

- Preprocess, remove comments/whitespace
- Expand pseudo-instructions
  - ex: 32-bit “li” becomes lui + ori
- Generate machine code for each instruction
  - Targets of jumps may be unknown at first pass
  - Fixed-length MIPS encoding helps here!
- Second pass to patch up jumps within file
  - Jumps to other files are still unknown for now

Linking

- Read all object files and static libraries
- Concatenate machine code from each
- Patch up cross-object jumps to correct address
- Add executable file headers
- Create import table for dynamic linker if needed

Compiler wrappers

- Most compilers (gcc etc) will preprocess, compile, assemble, and link with one command
- Use flags to dump middle steps for debugging
  - gcc -E stops after preprocessor
  - gcc -S stops before assembling
  - gcc -c stops before linking

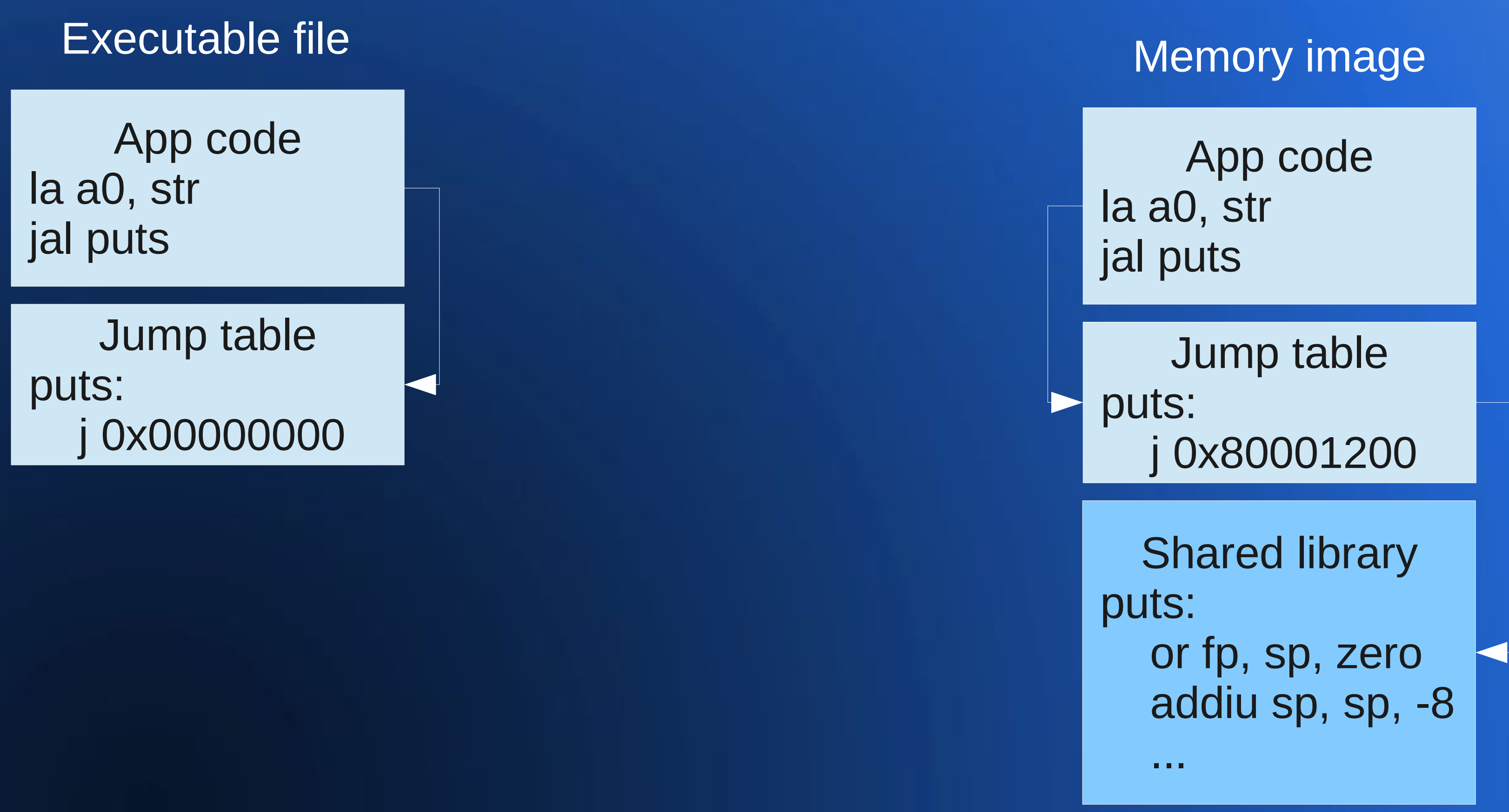
Problems with static linking

- Some code is used in lots of places
  - C standard library, etc
- Duplicating this code in every single program would waste lots of disk / memory space
- If a bug is found in the library, every program on the system would have to be recompiled!
- Is there a solution?

Dynamic linking

- Don't put the library code in the executable!
- Instead, just store a list of functions we need to grab from external libraries (DLL/SO)
- Linker creates jump table in code segment
- When OS loads executable, jump table is updated with pointers to actual library function
- Application code jumps to table entries

Dynamic linking



Disassembly

- Turn machine code back into asm
- Helpful for debugging compile (or sim) bugs
- objdump -d input_file
- Other useful flags
  - -D (disassemble everything)
  - -j .section_name (only disassemble one sec)
  - --source (show source if debug syms available)

Cross-compilation

- A compiler is just a program
- It runs on some particular CPU and OS
- It produces executables for some CPU and OS
- These two don't have to be the same!
- A compiler that targets a platform other than the one it runs on is called a *cross-compiler.*

Cross-compilation use cases

- Generating input for simulators
- Writing the first compiler for a new CPU
  - Solves chicken-and-egg problem
- Targeting a platform too small for a compiler
  - 8-bit MCU with 128 bytes RAM, etc
- Targeting a slow platform
  - Android phones run Linux and can run gcc just fine, but a desktop PC is much faster

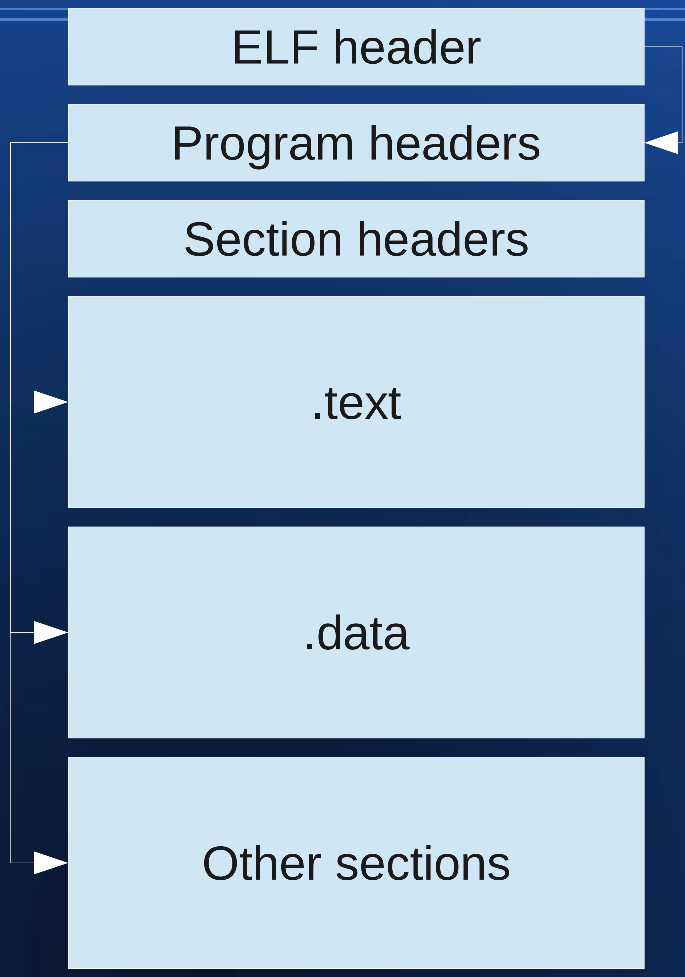
Executable files

- Contains machine code for program
- But that's not all! Also contains
  - Global data
  - Virtual memory addresses to put code at
  - Debug symbols (optional)
  - Symbol table for dynamic linker
  - Header info (where in the file to find the above)

Executable file formats

- PE (Portable Executable)
  - Windows EXE, DLL, CPL, SCR
- MACH-O
  - Mac OS X / iOS executables / dylib / o / core
- ELF (Executable and Linkable Format)
  - Linux executables / so / o / core
  - Simple, well documented, and supports MIPS
  - Your simulator will use this as the input format

ELF file format



Used for static/dynamic linking, we ignore this

May be >1 section in a memory segment

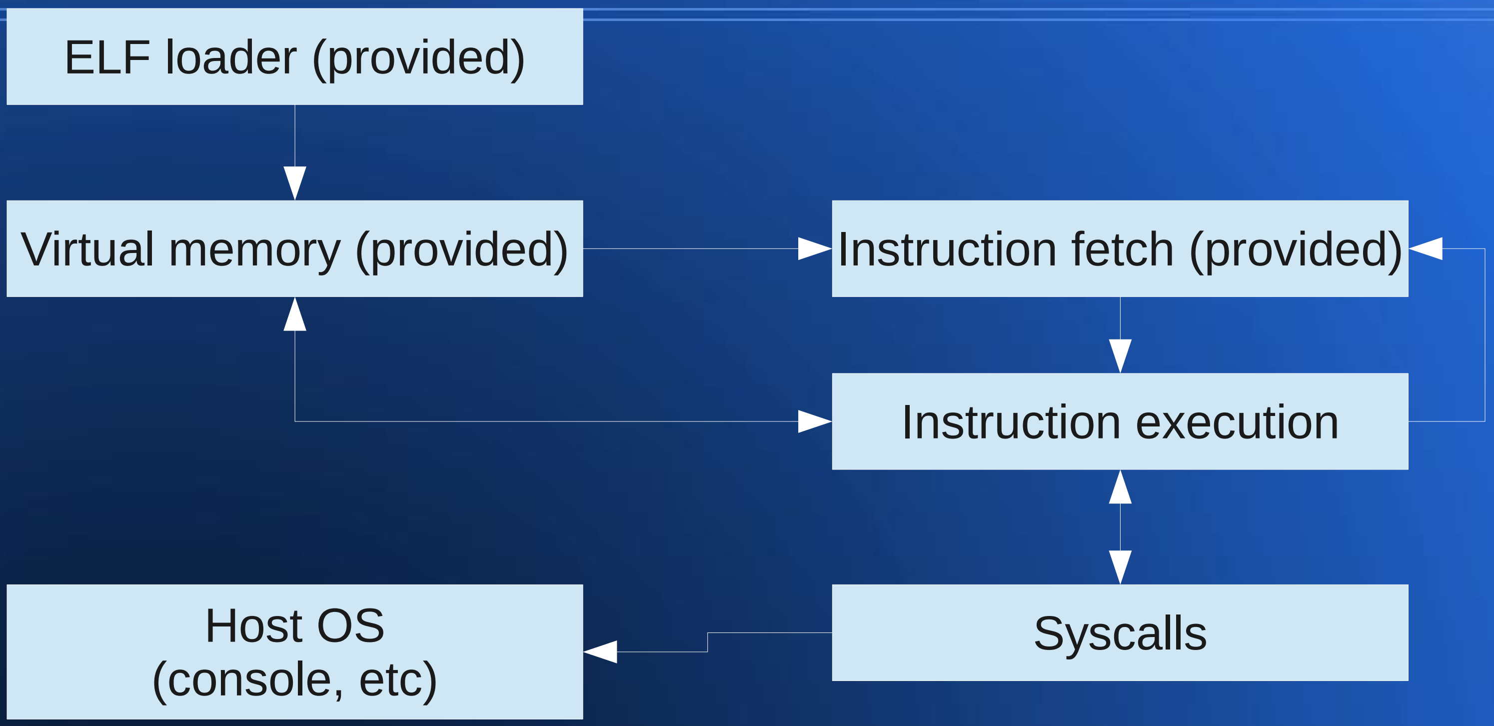
ELF header

- Signature (“this is an ELF file”)
- ELF class (32 or 64 bit OS)
- Endianness flag
- Version info, OS ABI, etc
- Type of file (executable, library, object, core)
- Instruction set
- Pointers to program/section headers

Program header

- A section is a contiguous region of virtual mem
- Each phdr describes one section
  - Type (PT_LOAD = belongs in mem)
  - Size in memory
  - Size on disk (may be zero for uninitialized data)
  - Virtual address of section
  - Pointer to contents in ELF file

Simulator structure



Example sim run

```
int main()
{
const char* str = "hello world\n";
do_syscall((unsigned int)str, 0, SYS_PRINT_STR);
return 0;
}
```

Reading ELF file ../c_testprog/c_testprog.elf...

Virtual address of entry point is 00400130

Starting simulation...

hello world

[0040013c] got SYS_EXIT

Delay slots

- The original MIPS CPUs had a 5-stage pipeline
- By the time we know we're jumping, the next instruction is already being decoded!
- If we flush the pipeline, we waste time
- MIPS fixed this by adding a *delay slot*
  - Instruction after a jump is *always* executed!
  - Applies to both conditional and unconditional
- SPIM typically hides these details from you

Example of delay slot usage

- 4000f8: 03e00008 jr ra
- 4000fc: 27bd0018 addiu sp,sp,24

Delay slots in your simulator

- MIPS gcc output assumes CPU has delay slot
- Your makefile includes “-fno-delayed-branch”
  - Instructs gcc to always fill delay slot with nop
  - Code behaves the same with/without delay slot
  - But may be less efficient on CPUs with one
- If you keep this flag, you do not need to implement delay slots in your simulator
  - If your simulator properly handles delay slots, you will get extra credit

Simulator code structure

- main.c - ELF loader and input
- sim.c - virtual memory layer, core of simulator

C bitfields

- Struct fields do not have to be mult of 1 byte
- unsigned int x:5; //5-bit long field
- Good for saving memory on embedded chips
- Also allows decoding of packed binary
  - Like MIPS instructions!
  - But how do we load the bitfields?

C unions

- Like a struct, but all members share the same block of memory
- Originally invented for letting several variables share memory in low-end systems
- But also allows easy casting!
  - Have a bitfield struct and an int as members

Unions for bitfield casting

```
union foo
{
    struct
    {
        unsigned int field1:5;
        unsigned int field2:19;
        unsigned int field3:8;
    } bar;
    uint32_t baz;
}
```

Demo

Questions?