ECE 471 – Embedded Systems
Lecture 6

Vince Weaver

http://www.eece.maine.edu/∼vweaver

vincent.weaver@maine.edu

18 September 2014

Announcements

- I have a cold and my voice is gone!
- HW#3 will be posted tomorrow

ARM Instruction Set Encodings

- ARM – 32 bit encoding
- THUMB – 16 bit encoding
- THUMB-2 – THUMB extended with 32-bit instructions
  - STM32L *only* has THUMB2
  - Raspberry PI *does not* have THUMB2
- THUMB-EE – some extensions for running in JIT runtime
- AARCH64 – 64 bit. Relatively new.

Recall the ARM32 encoding

`ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
cond				**0**	**0**	**0**	**0**	**1**	**0** **Opcode**	**0**	**S**	**Rn**			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				**Shift** **imm5**					**Shift** **typ**		**Sh** **Reg**	**Rm**			

THUMB

- Most instructions length 16-bit (a few 32-bit)
- Only r0-r7 accessible normally
  `add`, `cmp`, `mov` can access high regs
- Some operands (sp, lr, pc) implicit
  Can't always update sp or pc anymore.
- No prefix/conditional execution
- Only two arguments to opcodes
  (some exceptions for small constants: `add r0,r1,#1`)
- 8-bit constants rather than 12-bit

- Limited addressing modes: [rn,rm], [rn,#imm], [pc|sp,#imm]
- No shift parameter ALU instructions
- Makes assumptions about “S” setting flags (gas doesn’t let you superfluously set it, causing problems if you naively move code to THUMB-2)
- new push/pop instructions (subset of ldm/stm), neg (to negate), asr,lsl,lsr,ror, bic (logic bit clear)

THUMB/ARM interworking

- See `print_string_armthumb.s`
- BX/BLX instruction to switch mode.
  If target is a label, *always* switchmode
  If target is a register, low bit of 1 means THUMB, 0 means ARM
- Can also switch modes with `ldrm`, `ldm`, or pop with PC as a destination
  (on armv7 can enter with ALU op with PC destination)
- Can use `.thumb` directive, `.arm` for 32-bit.

THUMB-2

- Extension of THUMB to have both 16-bit and 32-bit instructions
- 32-bit instructions *not* standard 32-bit ARM instructions. It's a new encoding that allows an instruction to be 32-bit if needed.
- Most 32-bit ARM instructions have 32-bit THUMB-2 equivalents *except* ones that use conditional execution. The `it` instruction was added to handle this.
- `rsc` (reverse subtract with carry) removed

- Shifts in ALU instructions are by constant, cannot shift by register like in arm32
- THUMB-2 code can assemble to either ARM-32 or THUMB2

  The assembly language is compatible.

  Common code can be written and output changed at time of assembly.
- Instructions have “wide” and “narrow” encoding.

  Can force this (`add.w` vs `add.n`).
- Need to properly indicate “s” (set flags).

  On regular THUMB this is assumed.

THUMB-2 Coding

- See `test_thumb2.s`

- Use `.syntax unified` at beginning of code

- Use `.arm` or `.thumb` to specify mode

New THUMB-2 Instructions

- BFI – bit field insert
- RBIT – reverse bits
- movw/movt – 16 bit immediate loads
- TB – table branch
- IT (if/then)
- cbz – compare and branch if zero; only jumps forward

Thumb-2 12-bit immediates

```
top 4 bits 0000 -- 00000000 00000000 00000000 abcdefgh
           0001 -- 00000000 abcdefgh 00000000 abcdefgh
           0010 -- abcdefgh 00000000 abcdefgh 00000000
           0011 -- abcdefgh abcdefgh abcdefgh abcdefgh
           0100 -- 1bcdedfh 00000000 00000000 00000000
            ...
           1111 -- 00000000 00000000 00000001 bcdefgh0
```

Compiler

- RASPBERRY PI DOES NOT SUPPORT THUMB2

- `gcc -S hello_world.c`
  By default is arm32

- `gcc -S -march=armv5t -mthumb hello_world.c`
  Creates THUMB (won't work on Rapberry Pi due to HARDFP arch)

- `-mthumb -march=armv7-a` Creates THUMB2

IT (If/Then) Instruction

- Allows limited conditional execution in THUMB-2 mode.
- The directive is optional (and ignored in ARM32) the assembler can (in-theory) auto-generate the IT instruction
- Limit of 4 instructions

Example Code

```
it cc
addcc r1,r2
```

```
itete cc
addcc r1,r2
addcs r1,r2
addcc r1,r2
addcs r1,r2
```

ll Example Code

```
        ittt cs @ If CS Then Next plus CS for next 3
discrete_char:
        ldrbcs  r4,[r3]      @ load a byte
        addcs   r3,#1        @ increment pointer
        movcs   r6,#1        @ we set r6 to one so byte
        bcs.n   store_byte   @ and store it
offset_length:
```

AARCH64

- 32-bit fixed instruction encoding
- 31 64-bit GP registers (x0-x30), zero register (x30)
- PC is not a GP register
- only branches conditional
- no load/store multiple
- No thumb

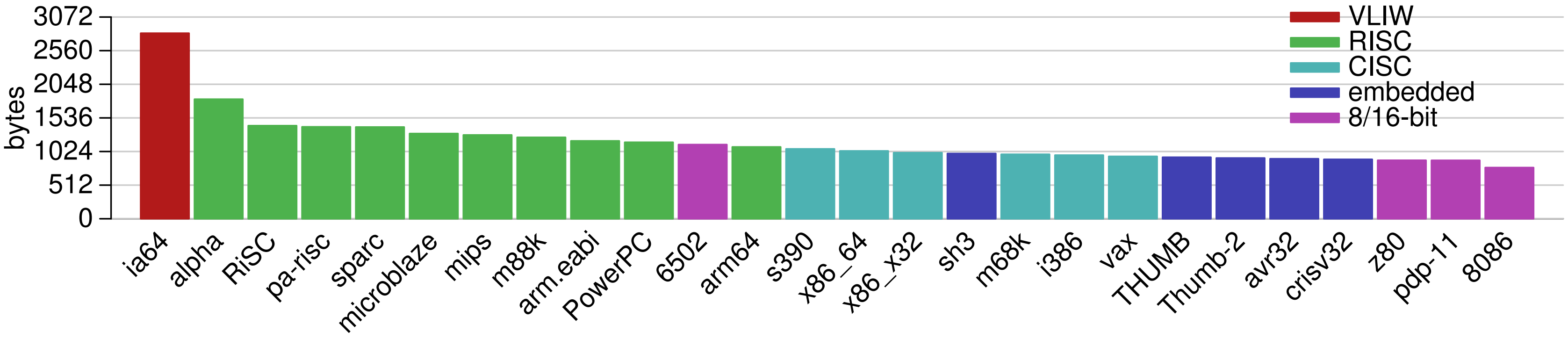
Code Density

- Overview from my `ll` ICCD'09 paper

- Show code density for variety of architectures, recently added Thumb-2 support.

- Shows overall size, though not a fair comparison due to operating system differences on non-Linux machines

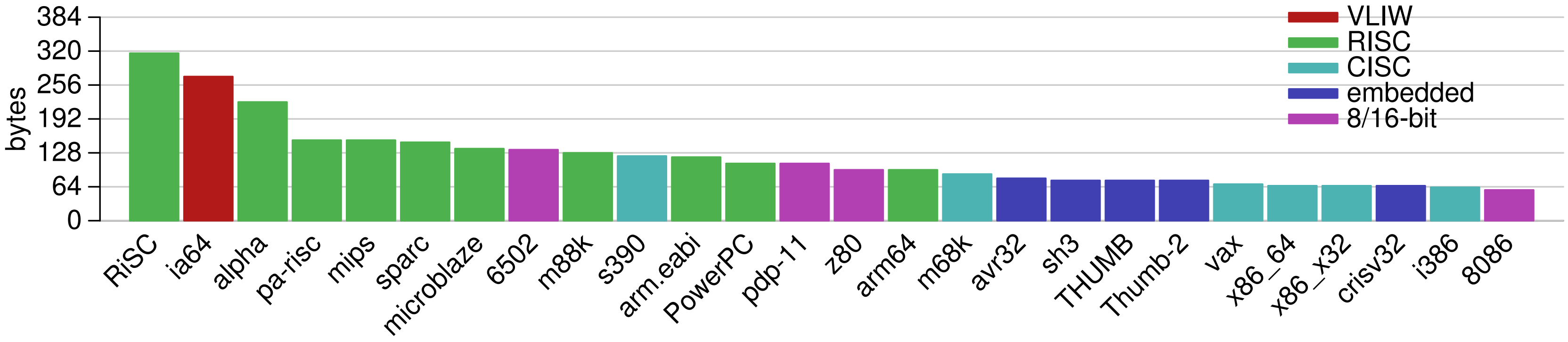
Code Density – overall



lzss compression

- Printing routine uses lzss compression
- Might be more representative of potential code density

Code Density – lzss



Low-Level ARM Linux Assembly

Kernel Programming ABIs

- OABI – “old” original ABI (arm). Being phased out. slightly different syscall mechanism, different alignment restrictions
- EABI – new “embedded” ABI (armel)
- hard float – EABI compiled with ARMv7 and VFP (vector floating point) support (armhf). Raspberry Pi (raspbian) is compiled for ARMv6 armhf.

System Calls (EABI)

- System call number in r7
- Arguments in r0 - r6
- Call `swi 0x0`
- System call numbers can be found in `/usr/include/arm-linux-gnueabihf/asm/unistd.h` They are similar to the 32-bit x86 ones.

System Calls (OABI)

The previous implementation had the same system call numbers, but instead of r7 the number was the argument to `swi`. This was very slow, as there is no way to determine that value without having the kernel backtrace the callstack and disassemble the instruction.

Manpage

The easiest place to get system call documentation.

```
man open 2
```

Finds the documentation for “open”. The 2 means look for system call documentation (which is type 2).

A first ARM assembly program: `hello_exit`

```
.equ SYSCALL_EXIT,     1

        .globl _start
_start:

        #================================
        # Exit
        #================================
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT        @ put exit syscall number (1) in eax
        swi     0x0                     @ and exit
```

hello_exit example

Assembling/Linking using make, running, and checking the output.

```
lecture6$ make hello_exit_arm
as -o hello_exit_arm.o hello_exit_arm.s
ld -o hello_exit_arm hello_exit_arm.o
lecture6$ ./hello_exit_arm
lecture6$ echo $?
5
```

Assembly

- @ is the comment character. # can be used on line by itself but will confuse assembler if on line with code. Can also use /* */

- Order is source, destination

- Constant value indicated by # or $

Let's look at our executable

- `ls -la ./hello_exit_arm`
  Check the size

- `readelf -a ./hello_exit_arm`
  Look at the ELF executable layout

- `objdump --disassemble-all ./hello_exit_arm`
  See the machine code we generated

- `strace ./hello_exit_arm`
  Trace the system calls as they happen.

hello_world example

```
.equ SYSCALL_EXIT,     1
.equ SYSCALL_WRITE,    4
.equ STDOUT,           1

        .globl _start
_start:
        mov     r0,#STDOUT              /* stdout */
        ldr     r1,=hello
        mov     r2,#13                  @ length
        mov     r7,#SYSCALL_WRITE
        swi     0x0

        # Exit
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT        @ put exit syscall number in r7
        swi     0x0                     @ and exit

.data
hello:          .ascii "Hello␣World!\n"
```

New things to note in `hello_world`

- The fixed-length 32-bit ARM cannot hold a full 32-bit immediate
- Therefore a 32-bit address cannot be loaded in a single instruction
- In this case the “=” is used to request the address be stored in a “literal” pool which can be reached by PC-offset, with an extra layer of indirection.

Put string example

```
.equ SYSCALL_EXIT,     1
.equ SYSCALL_WRITE,    4
.equ STDOUT,           1

        .globl _start
_start:
        ldr     r1,=hello
        bl      print_string            @ Print Hello World
        ldr     r1,=mystery
        bl      print_string            @
        ldr     r1,=goodbye
        bl      print_string            /* Print Goodbye */

        #================================
        # Exit
        #================================
exit:
        mov     r0,#5
        mov     r7,#SYSCALL_EXIT        @ put exit syscall number (1) in eax
        swi     0x0                     @ and exit
        #====================
        # print string
        #====================
        # Null-terminated string to print pointed to by r1
        # r1 is trashed by this routine

print_string:
        push    {r0,r2,r7,r10}          @ Save r0,r2,r7,r10 on stack

        mov     r2,#0                   @ Clear Count

count_loop:
        add     r2,r2,#1                @ increment count
        ldrb    r10,[r1,r2]             @ load byte from address r1+r2
        cmp     r10,#0                  @ Compare against 0
        bne     count_loop              @ if not 0, loop

        mov     r0,#STDOUT              @ Print to stdout
        mov     r7,#SYSCALL_WRITE       @ Load syscall number
        swi     0x0                     @ System call

        pop     {r0,r2,r7,r10}          @ pop r0,r2,r7,r10 from stack

        mov     pc,lr                   @ Return to address stored in
```

```
                                         @ Link register

.data
hello:          .string "Hello␣World!\n"    @ includes null at end
mystery:        .byte 63,0x3f,63,10,0       @ mystery string
goodbye:        .string "Goodbye!\n"        @ includes null at end
```

Clarification of Assembler Syntax

- @ is the comment character. # can be used on line by itself but will confuse assembler if on line with code. Can also use /* */

- Constant value indicated by # or $

- Optionally put % in front of register name