

# ECE 471 – Embedded Systems

## Lecture 10

Vince Weaver

`https://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

20 September 2023

# Announcements

- Don't forget HW#3
- Note on issues if running fully 64-bit version of OS  
Posted some directions, let me know if you still need help  
Next year should probably provide 64-bit code



# HW2 Review – Code

- Most people seem to be accessing the Pi OK
- Be sure to follow directions!
- Put your name in the README
- Testing. How can you test? `wc -l`
- Watch for off-by-one errors
- Comment your code!
- Also watch out for compiler warnings! (Though each compiler version might have different warnings)
- **Error handling! especially for command line parsing**



- Most C code OK.

Be sure if it says print 15 lines that you do, not 16.

Colors seem not to be a problem.



# HW2 Review – Questions

- more info on ls. Looking for man. “info” or `ls --help`
- `ls -a` shows hidden files. Hidden files on UNIX
- Why use C? close to hardware, easier than assembly, etc. careful with “faster”. It often is, but give reasoning and what you are comparing (faster than python? asm?)



# Really Brief Overview of ARM32 Assembler

- Processor has 16 registers
  - r0.. r12
  - SP (stack pointer) (points to LIFO stack)
  - LR (link register) when you branch and link to a function, gets the value you are branching from so you can return to it  
how do you handle non-leaf functions? (save on stack usually)
  - PC (program counter) on original ARM design they



were clever and had this as a normal register that you can operate on with regular instructions. In practice not the best ide

- Instructions have three arguments, destination and two sources

add r0,r1,r2 means  $r0 = r1 + r2$



# ARM32 Instruction Types

- There are many, possibly hundreds
- math: add, sub, mul, div
- bitwise logical: and, orr, eor, bic
- loads/stores: ldr, str
- compare: cmp
- branch: b, beq, bne, bl, blx
- floating point, other extensions





# Weird ARM32 Things – Barrel Shifter

- Most ALU instructions you can include a shift/rotate for free
- `add r0,r0,r0 LSL #4`  
Sneaky way to multiply by 5
- Helps with code density



# Weird ARM32 Things – Constants

- Loading constant values into registers is hard
- Can use the = trick for 32-bit, as you can't fit a 32-bit constant into a 32-bit instruction
- Regular constants are 12-bits but there are various tricks to make it more useful and not just a flat 0..4095 value



# Weird ARM32 Things – Weird Addressing modes

- ldr, ldrb, ldrsb, etc
- `ldrb r1, [r2]` – load value in address/pointer in r2
- `ldrb r1, [r2, #20]` – offset  $r2 +$  value, useful in structs
- `ldrb r1, [r2, r3]` – add  $r2 + r3$  for address
- `ldrb r1, [r2, r3, LSL #2]` – shift before adding, useful array index
- `ldrb r1, [r2, #20]!` – load from  $r2 + 20$ , then update address into r2



- `ldrb r1, [r2], #20` – load from r2, then an and update address into r2



# ARM32 Conditional Execution

Why are branches bad?

```
if ( x == 5 )
```

```
    a+=2;
```

```
else
```

```
    b-=2;
```

```
        cmp        r1 , #5
```

```
        bne        else
```

```
        add        r2 , r2 , #2
```

```
        b          done
```

```
else :
```

```
        sub        r3 , r3 , #2
```

```
done :
```

@ equivalent w/o branches



```
cmp    r1 , #5
addeq  r2 , r2 , #2
subne  r3 , r3 , #2
```



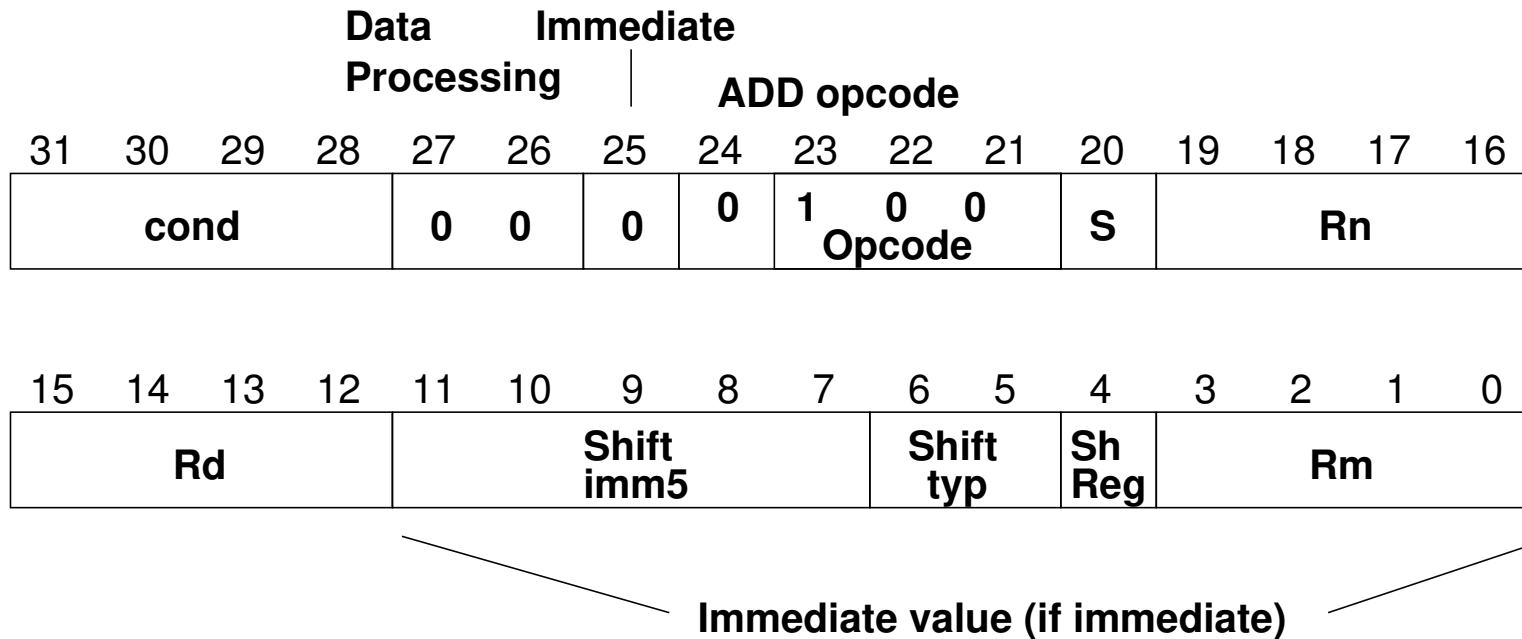
# ARM Instruction Set Encodings

- ARM – 32 bit encoding
- THUMB – 16 bit encoding
- THUMB-2 – THUMB extended with 32-bit instructions
  - STM32L *only* has THUMB2
  - Original Raspberry Pis *do not* have THUMB2
  - Raspberry Pi 2/3 *does* have THUMB2
- THUMB-EE – extensions for running in JIT runtime
- AARCH64 – 64 bit. Relatively new. Completely different from ARM32



# Recall the ARM32 encoding

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





# Why Code Density

- Smaller code can be better
- Lower resources: Cheaper? If you can fit more features into smaller RAM or disk you can save money
- Faster? It depends. Modern chips are really hard to predict, but if your processor has Caches and you can fit better in instruction cache it can potentially speed things up a lot



# 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.  
Sets/clears the T (thumb) flag in status register  
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
- The 32-bit instructions are *not* the standard 32-bit ARM instructions.
- 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
- Most cannot have PC as src/dest



- 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
rotate bottom 7 bits|0x80 right by top 5 bits
             01000 -- 1bcdefgh 00000000 00000000 00000000
             ...
             11111 -- 00000000 00000000 00000001 bcdefgh0
```



# Compiler

- Original 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 Raspberry Pi due to HARDFP arch)
- `-mthumb -march=armv7-a+fp` Creates THUMB2 for



pi



# 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
```

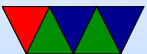
```
itete cc
```

```
addcc r1,r2
```

```
addcs r1,r2
```

```
addcc r1,r2
```

```
addcs r1,r2
```



# 11 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
- 32 64-bit GP registers
  - x0 - x7 = args
  - x8 - x18 = temp (x8=syscall num during syscall)
  - x19-x28 = callee saved
  - x29 = frame pointer
  - x30 = link register
  - x31 = zero register or stack pointer
- PC is not a GP register





- only branches conditional
- no load/store multiple
- No thumb

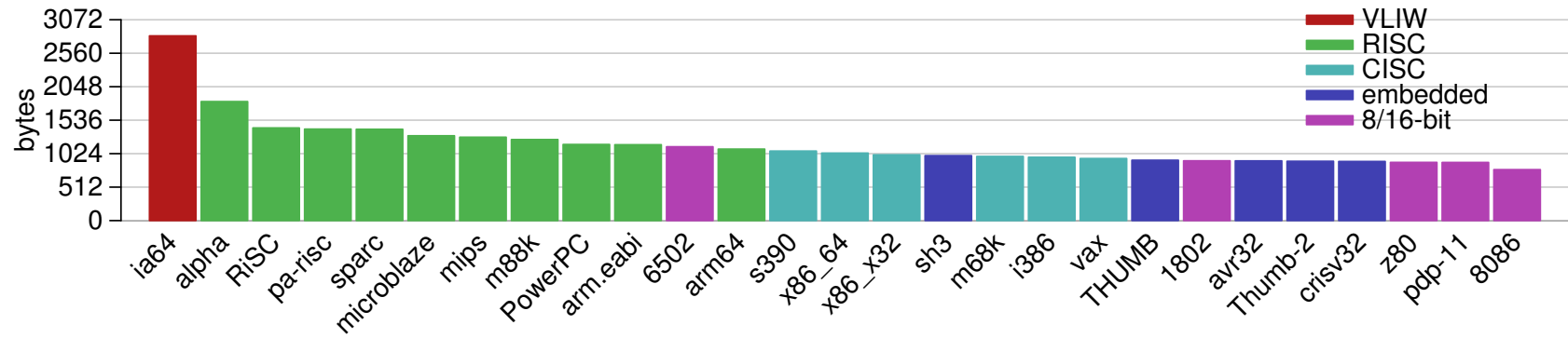


# Code Density

- Overview from my 11 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

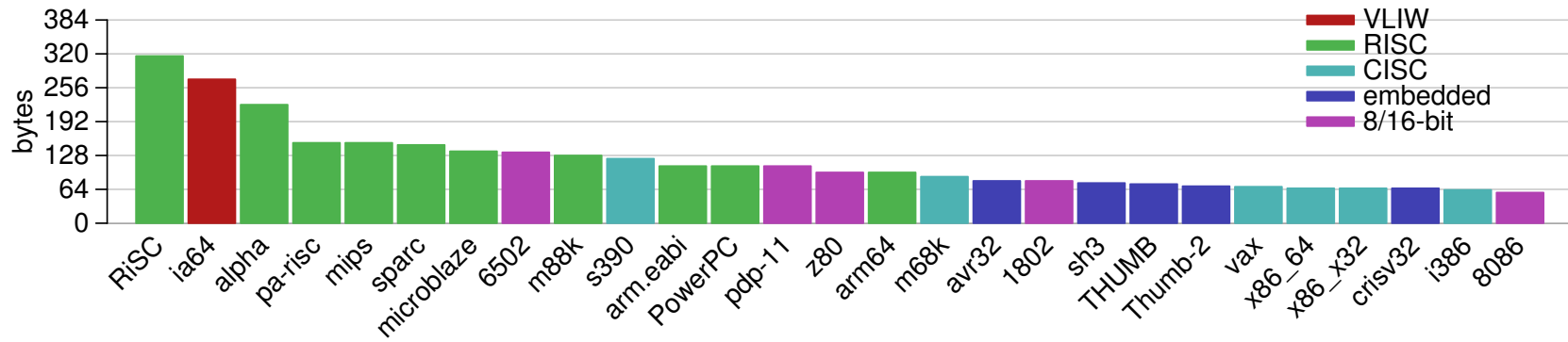


# lzss compression

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



# Code Density – Izss



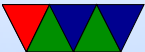
# Put string example

```
.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
```

