# cse30 discussion 5

Ibrahim Awwal

July 13, 2015

# raspberry pi setup redux

# port forwarding

- Makes a local port available on the internet on a different port (not needed from UCSD-PROTECTED)
- This means you can ssh into your Raspberry Pi from outside your LAN
- Run `rpi_upnp.sh` from **github.com/ibrahima/raspi_networking**
  - Uses UPnP to automatically open an external port on your router
  - Default port is 10022
- `ssh pi@your-public-ip -p 10022`

# dynamic dns

- Lets you set up a custom domain name for your public IP address (which could change)
- DuckDNS is a free no-nonsense service provider, feel free to use any other
- Detailed instructions are **on github**
- Once you've set this up, you can then do `ssh pi@you.duckdns.org -p 10022`

# ssh configuration

- Can store commonly used hosts in `~/.ssh/config` (Linux/Mac)

```
Host rpi1
     Hostname me.duckdns.org
     Port 10022
     User pi
```

- If you **set up an SSH key** without a passphrase or use `ssh-agent`, you can avoid typing your password too
- Now you can just type `ssh rpi1` and log in immediately!

# any questions?

# programming assignment 2

# tips

- Come to lab hours
- If you're having issues with your Raspberry Pi, do the C parts first on ieng6 or your own computer
- If you get segfaults, run your program through gdb, get a backtrace
- Make sure to compile with −g for debug symbols

# tips

- The array is sorted, so take advantage of this when appropriate
- Related to the above, make sure to keep the array sorted
- Make sure variables that need to live past the life of a function are heap allocated
- Any questions?

# arm assembly

# assembly language

- Assembly language is a 1-to-1 mapping to machine code
- Instructions are basically mnemonics for the programmer to refer to binary
- *Assembler* is the program that turns these mnemonics into machine code
- Instructions operate on *registers*, small memory directly within the CPU

# instruction types

- Arithmetic: Only processor and registers involved
- Data Transfer Instructions: Interacts with memory
- Control Transfer Instructions: Change flow of execution
- examples of each?
- **ISA Quick Reference Card**

# arithmetic

- `ADD dest, op1, op2` - op2 can be a constant (immediate) encoded in the instruction
  - What's one way to copy the contents of one register into another?
- `MUL`/`SMULL` - what's the difference?
- Shifting: `LSL`, `LSR`, `ASL`, `ASR`
  - No instruction of their own, combine with other instruction (eg. `ADD`/`MOV`)
  - When might we use this instead of `MUL`? Why?
- Be aware of data sizes, sign, overflow bits
- `QADD`, `ADDS`, diff . . .
- **Reference on ARM instruction timing**

# data transfer

- LDR loads a word from memory into a register (4 bytes)
- STR stores a word from register into memory
- `LDR dest, [base #offset]`
- many options for calculating offset, updating the base register, pre-indexing vs post-indexing
- Why do we have all these options?

# control flow

- Branch instructions change the program counter (instead of incrementing by 1 instruction)
  - `B` **b**ranches to a label
  - `BL` **b**ranches to a label and **l**inks return address into `LR`
  - `BX` branches to a register*
- Most instructions in ARM can be conditionally executed, not just branches!
- eg. `CMP r4, #0`; `BEQ *label*` –> `if(r4 != 0){}`
- What is the benefit of conditional execution?

# comparisons

- `CMP r1, r2` or `CMP r1, #immediate` –> `r1 - r2`
- Stores result of comparison in status bits
  - N: Negative
  - Z: Zero
  - C: Carry (or Unsigned Overflow)
  - V: (Signed) Overflow
  - Status bits are also used when doing arithmetic with overflows
- Add a condition code to any instruction to execute it conditionally
  - eg. `EQ`, `NE`, `GE`, `LT`, etc.
- **Reference on condition codes and status bits**

# generating assembly from c

- `gcc source.c -S` will output source.s assembly
- `gcc -c -g -Wa,-a,-ad source.c > source.lst` will output a mixed C/assembly listing
- latter command taken from **`http://www.delorie.com/djgpp/v2faq/faq8_20.html`**
- tip: save these as aliases in your shell (eg. .bashrc)
    - `alias asmc="gcc -c -g -Wa,-a,-ad"`
    - use it like `asmc source.c > source.lst`

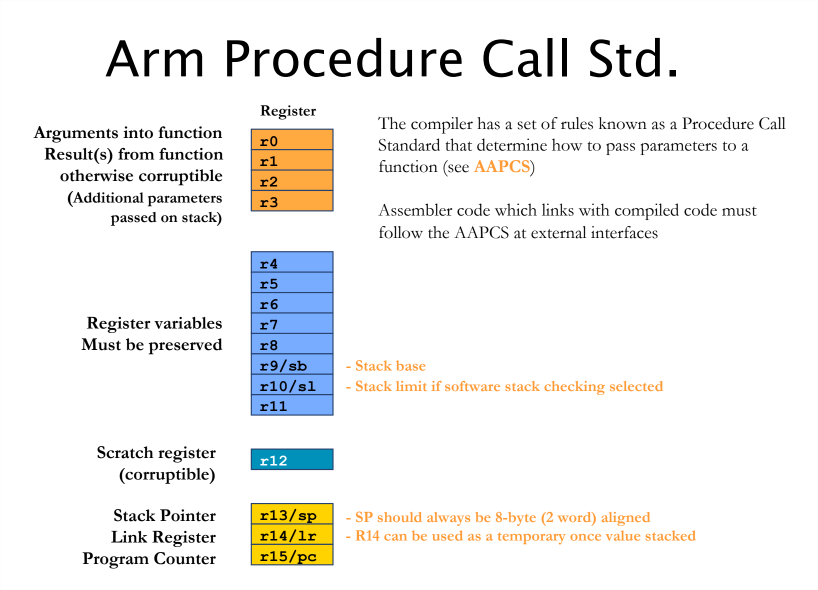
# function calls

- To call a function, we need to transfer execution to a different location in code, with some arguments passed and a return value received
    - Transfer execution: `B` to a label
    - Function arguments: passed in r0-r3, more on stack
    - What about longs/doubles?
    - Return value is put into r0
- How do we get back to our code?

# function calls

- We use the `BL` instruction to store the return address (next instruction) into `LR`
- Function call then returns by performing `BX LR`
- Function signatures are a contract that both callee and caller must obey
- Registers must be preserved across function calls
- **ARM Architecture Procedure Call Standard (AAPCS)**

# diagram from lecture



## Arm Procedure Call Std.

Register

Arguments into function
Result(s) from function
otherwise corruptible
(Additional parameters
passed on stack)

r0
r1
r2
r3

The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see **AAPCS**)

Assembler code which links with compiled code must follow the AAPCS at external interfaces

Register variables
Must be preserved

r4
r5
r6
r7
r8
r9/sb - Stack base
r10/sl - Stack limit if software stack checking selected
r11

Scratch register
(corruptible)

r12

Stack Pointer
Link Register
Program Counter

r13/sp - SP should always be 8-byte (2 word) aligned
r14/lr - R14 can be used as a temporary once value stacked
r15/pc

# pa2 starter code

```
.func get_min_ARM, get_min_ARM
.type get_min_ARM, %function

get_min_ARM:
    push {r4-r11, ip, lr} @ Save caller's registers on the
    @ put your return value in r0
return:
    pop {r4-r11, ip, lr} @ restore caller's registers
    BX lr
.endfunc

.end
```

# examples

```
int fun(int a, int b, int c, int d, int e){
    return a + b - c + d - e;
}
```

- Where does e go?

# examples

```
int fun(int a, int b, int c, int d, int e){
    return a + b - c + d - e;
}
```

- Where does e go?

```
long fun(int a, int b){
    return (long)a*(long)b;
}
```

- Where do we store the return value?

# tips

- Before you write, plan out detailed pseudocode
- Comment **every** line, with detailed overview for functions
- Trace execution, draw registers and how they change, how PC changes, etc

# translating c language constructs to assembly

# flow control

- if –> conditional branch, skip over some instructions
- loops –> jump back to start of loop if condition satisfied

# accessing arrays, structs, pointers

- To index arrays, add index*sizeof(type) to base register
    - eg. `LDR r2, [r3 r4]` –> `r2 = r3[r4]`
    - what type could array r3 be?
    - what if we have an int array?
- Struct fields are laid out sequentially in memory, aligned based on their size
    - **Important**: memory layout is aligned to size of variable
    - This means if you have a struct with char and int, the first field will be padded so that the second starts on a word boundary
    - **Interesting article on struct packing**
    - The `offsetof()` macro can tell you the offset (in bytes) to a field of a struct
- Pointers are dereferenced in the same way as arrays - `LDR r2, [r3]` <–> r2=*r3

# arm assembly exercises

# understanding instructions

- `ADD8 r0, r1, r2`
- `CMP r3, r4`
- `MOVEQ r2, #-1`
- `LDR r5, [r6, r7 LSL #4]`

# writing arm: exponentiation

Returns x^y (^ is bitwise XOR in C)

```
int exp(int x, int y);
```

# writing arm: strcpy

```
char * strcpy ( char * destination, const char * source );
```

# fun note: tis-100

- There's an actual assembly language programming game called **TIS-100** on Steam
- Haven't tried it but it could be fun and educational