CEG2400 - Microcomputer Systems

Lecture 10: Subroutine Calls, DMA and ARM Architecture

Philip Leong

Introduction

- There are several ways to pass arguments to functions
  - Global variables
  - Registers
  - Stack
- ARM has defined a convention call the “ARM Architecture Procedure Call Standard” (AAPCS) so that separately compiled subroutines can call each other
- We will only discuss a simplified, ARM (not thumb) subset of this standard
- Main reference: http://www.arm.com/pdfs/aapcs.pdf

Core Registers and Usage

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

Function Calls

- r0-r3
  - pass arguments to a subroutine. Arguments are allocated first to registers and excess arguments are placed on the stack
  - return a value from a function
  - used to store temporary values between subroutine calls i.e. scratch registers

Register variables

- r4-r11 are v0-v8 holds a routine’s local variables
  - Actually, r9 is platform specific we assume it is used for v6
  - Subroutines must preserve these registers (if modified)

ip register

- Can be used as a scratch register or to aid in long jumps

Examples

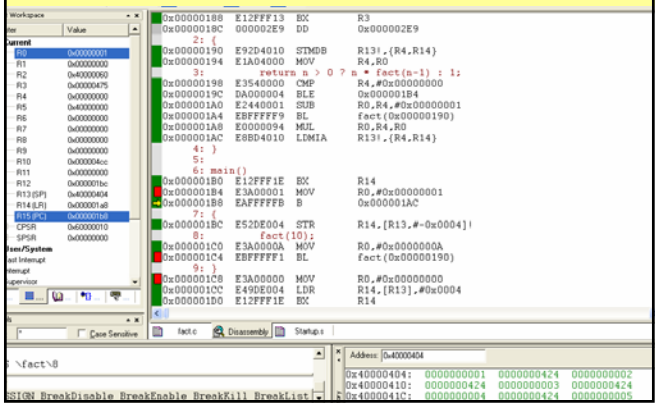
- Revisit last week's examples
  - Ex 5: more than 4 arguments to a function
  - Ex 6: have to save lr on stack in nested calls

A recursive function

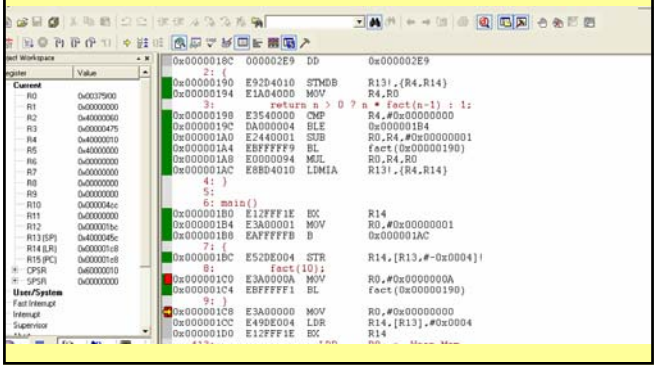
```
/* computes factorial */
fact(int n)
{
   return n > 0 ?
        n * fact(n-1) : 1;
}

main()
{
   fact(10);
}
```


Disassembly and Stack



Where is the answer?



Quiz

- Fibonacci numbers are described by the sequence
  - F(n)=F(n-1)+F(n-2) with F(1)=1 and F(0)=0
  - Write an ARM assembly language conforming to AAPCS that computes F(n) (n > 0)

Fib in C

```
fib(unsigned n)
{
   if (n == 0)
        return 0;
   else if (n == 1)
        return 1;
   else
        return fib(n-1)+fib(n-
   2);
}
```

- First
  - How are we going to organize the registers
  - r4 and r5 store fib(n-1) and fib(n-2) respectively
  - Need to save on stack
  - Input comes in r0

Hand vs C

```
        stmdb   sp!,{r4,r5,lr}
        mov     r4,r0       ; save n
        cmp     r4,#0       ; =0?
        bne     ret0
        mov     r0,#0       ; not needed
ret     ldmia   sp!,{r4,r5,lr}
        bx      lr
ret0    cmp     r4,#1       ; =1?
        bne     fib1
        mov     r0,#1       ; not needed
        b       ret
fib1    sub     r0,r4,#1
        bl      fib
        mov     r5,r0
        sub     r0,r4,#2
        bl      fib
        add     r0,r0,r5
        b       ret
```

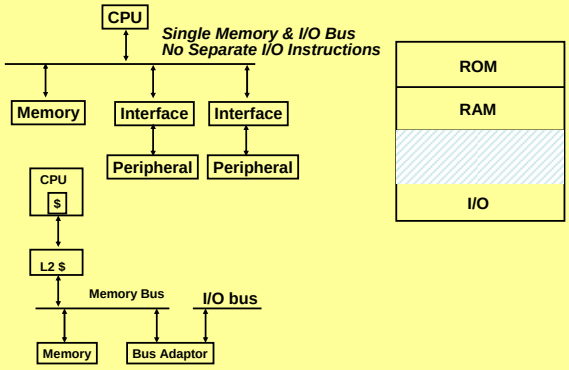
```
fib     cmp     r0,#1
        bxls    lr
        stmfd sp!,{r4,r5,lr}
        mov     r5,r0       ; save n
        sub     r0,r5,#1    ; compute n-1
        bl      fib         ; r0=fib(n-1)
        mov     r4,r0       ; r4=r0
        sub     r0,r5,#2    ; compute n-2
        bl      fib         ; r0=fib(n-2)
        add     r0,r4,r0    ; r0=fib(n-1)+fib(n-2)
        ldmfd sp!,{r4,r5,pc}
```

- Remove redundant operations e.g. fib(0)=0, fib(1)=1
- For case of 0 and 1, don't save stack
- Use conditional execution
- Organize code so there are fewer branches

Memory and DMA

Some slides courtesy Kurt Keutzer, UCB

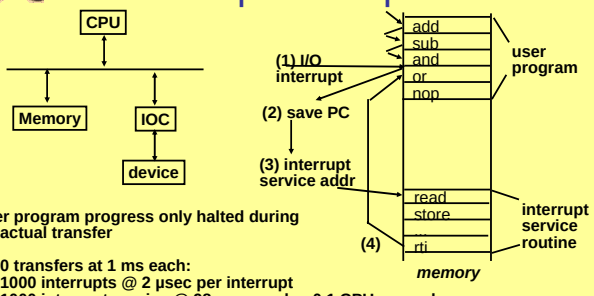
Memory Mapped Devices



Interrupts

- Advantage:
  - User program progress is only halted during actual transfer
- Disadvantage, special hardware is needed to:
  - Cause an interrupt (I/O device)
  - Detect an interrupt (processor)
  - Save the proper states to resume after the interrupt (processor)
  - Processor has overhead of entering and exiting interrupt

Interrupt Example



User program progress only halted during actual transfer

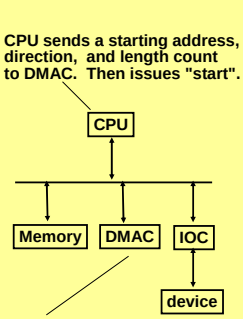
1000 transfers at 1 ms each:
1000 interrupts @ 2 µsec per interrupt
1000 interrupt service @ 98 µsec each = 0.1 CPU seconds

Device xfer rate = 10 MBytes/sec => 0 .1 x 10^{-6} sec/byte => 0.1 µsec/byte
=> 1000 bytes = 100 µsec
1000 transfers x 100 µsecs = 100 ms = 0.1 CPU seconds

Still far from device transfer rate! 1/2 in interrupt overhead

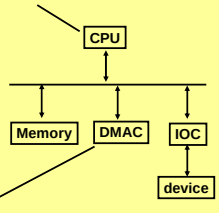
DMA

- Direct Memory Access (DMA):
  - External to the CPU
  - Act as a master on the bus
  - Transfers blocks of data to or from memory without CPU intervention



DMA Example

CPU sends a starting address, direction, and length count to DMAC. Then issues "start".

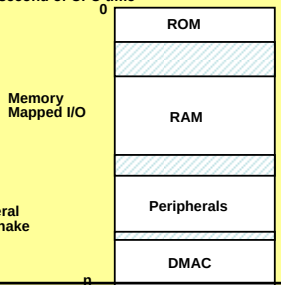


DMAC provides handshake signals for Peripheral Controller, and Memory Addresses and handshake signals for Memory.

Time to do 1000 xfers at 1 msec each:

1 DMA set-up sequence @ 50 µsec
1 interrupt @ 2 µsec
1 interrupt service sequence @ 48 µsec

.0001 second of CPU time



ARM Processor Organization

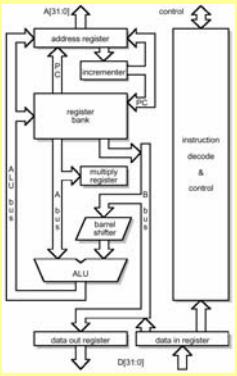
Slides Peter Cheung, Imperial College

History

- First ARM processor developed on 3 micron technology in '83-'85
- This course is mainly based on the ARM6/7 architecture developed between '90-'95.
- Digital Equipment Corporation (now Compaq) developed the StrongARM processor which is very high performance.
- Recent developments are: ARM8 and ARM9E (1999), and a ARM processor without clock - the asynchronous AMULET from U. of Manchester

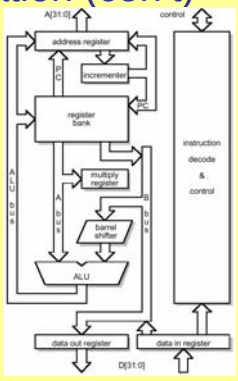
Internal Organization

- Two main blocks: datapath and decoder
- Register bank (r0 to r15)
  - Two read ports to A-bus/B-bus
  - One write port from ALU-bus
  - Additional read/write ports for program counter r15
- Barrel shifter - shift/rotate 2nd operand by any number of bits
- ALU performs arithmetic/logic functions
- Address registers/incrementer holds either PC address (with increment) or operand address



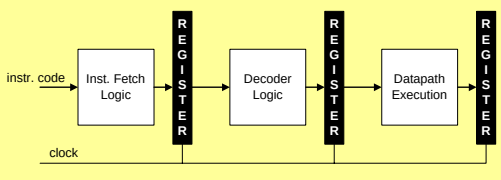
Internal Organization (con't)

- Data register holds read/write data from/to memory
- Instruction decoder decodes machine code instructions to produce control signals to datapath
- In single-cycle data processing instructions, data values are read on the A-bus & B-bus, the results from ALU is written back into register bank
- PC value in address register is incremented and copied back to r15 and the address register - this allows fetching new instructions ahead of time (instruction pre-fetch)

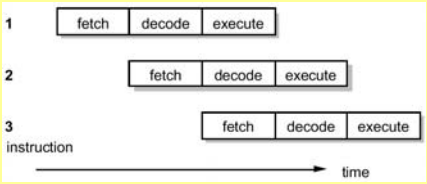


Pipelining

- ARM uses a 3-stage instruction pipeline
  - **Fetch**: **fetch instruction code from memory into the instruction pipeline**
  - **Decode: instruction decoded to obtain control signals for the datapath ready for the next stage**
  - **Execute: instruction "owns" the datapath - register read; shifting; ALU results generated and write-back**
- Results for each stage stored in registers
- The consequence is that the clock period is much shorter than without pipelining



Pipelining (con't)



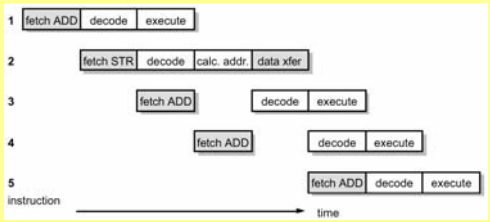
- At any time, 3 different instructions may occupy each of the the 3-stages of pipeline
- It may take three cycles to complete a single-cycle instruction. This is said to have a three cycle **latency**
- Once a pipeline fills, the processor completes a single-cycle instruction every clock cycle. Therefore the **throughput** is one instruction per cycle.

Pipeline of a multi-cycle instruction

- Consider this code sequence:

```
ADD   r1, r2, r3, LSL #3
STR   r0, [r1, #4] !          ; mem32[r1+4] := r0
ADD   r0, r0, r4              ;  ..... r1 := r+ 4
ADD   .....
ADD   ....
```

- The pipeline behaviour of this code sequence is:



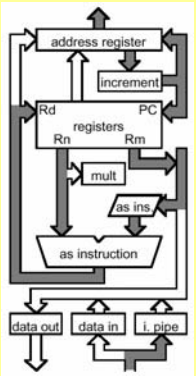
Pipeline of a multi-cycle instruction (con't)

- STR instruction is a two cycle instruction
- It has **four** cycle latency: fetch, decode, address calculation and data transfer
- The last two pipeline cycles are both execute phase
- Decode cycle always happens just BEFORE the execute phase
- During the first datapath cycle, each instruction issues a fetch for the **next instruction but one (i.e. two ahead)**
- Branch instruction poses a problem with pipeline because branch instruction may cause a change of program flow. Therefore the future instruction inside the pipeline may be wrong.
- This is called pipeline **control hazard**
- Solution: if there is a change of program flow, **flush the pipeline**!

Datapath activity during data processing instruction

```
ADDS  r0, r1, r2   LSL #3    ; r0 := r1 + r2 * 8
```

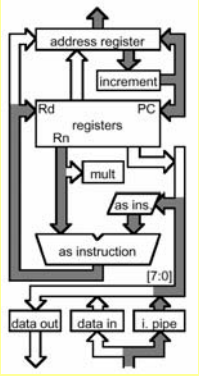
- Always requires two operands, in this case one from r1 and second from r2
- Second operand passes through barrel shifter
- ALU operate on operands and write results back to register r0
- Condition code register is updated (because the S-bit is set)
- PC value in address register is incremented and copied back to r15 & address register
- Next instruction but one (i.e. next-next instruction) is loaded into bottom of instruction pipeline (i.pipe)



Datapath activity during data processing instruction

```
SUB   r0, r1, #128   LSL #3   ; r0 := r1 - 128*8
```

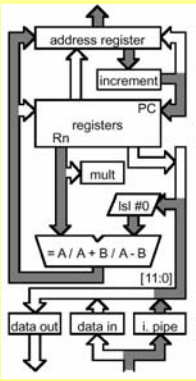
- Almost the same as previous instruction except that the bottom 8-bits of the instruction is used to provide the second operand
- It is shifted via the barrel shifter before processing



Data Transfer Instructions – 1st cycle

```
STR   r0, [r1, #4]!          ; mem32[r1+4] := r0
                             ; r1 := r1 + 4
```

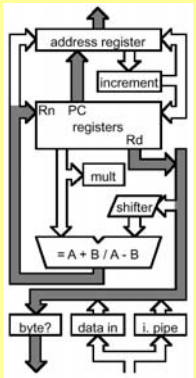
- ALU is used to compute the "effective address", i.e. the address in memory for storing
- r1 is read to ALU, #4 is obtained from instruction (bottom 12-bit offset value) and r1+4 is computed
- The store address is sent to address register ready for the next cycle



Data Transfer Instructions – 2nd cycle

```
STR    r0, [r1, #4]!        ; mem32[r1+4] := r0
                            ; r1 := r1 + 4
```

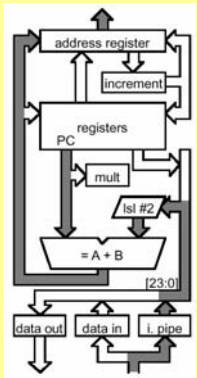
- During the second cycle, r0 is written out to data bus
- If it is a STRB (store-byte) instruction, the bottom 8-bit of r0 is replicated 4 times across the 32-bit data bus, a external logic selects the appropriate byte to write
- During this cycle, the new address value computed in the previous cycle is used to update r1 (because of '!')



Branch Instructions – 1st Cycle

```
BL     sub1     ; branch and link to sub1
```

- This is a three cycle instruction. During 1st cycle, 24-bit immediate (i.e. offset value) is extracted from the instruction
- It is left shifted by 2 bits to give a word-aligned offset
- It is then added with the PC value to form the destination address stored in address register



Branch Instructions – 2nd & 3rd Cycles

```
BL     sub1     ; branch and link to sub1
```

- During the 2nd cycle, the link register is updated with the return address
- The new instruction is fetched to fill the instruction pipeline and the PC is updated
- A third cycle is also needed for the new instruction to progress through the pipeline. While this is happening, the value stored in the link register is also corrected so that it is pointing to pc+4 instead of pc+8.