SPRING 2016

CMPE 364

Microprocessor Based Design

Dr. Mohamed Al-Meer

Lecture Expectations

- Expected to achieve:
  - Writing assembly code
  - Cycle Counting
  - Instruction scheduling
    - Load instructions
    - Preloading
    - Loop unrolling
  - Register Allocation
    - Allocating variables to registers
    - Making the most of available registers

Writing Assembly Code

- Embedded software projects contain a few key subroutines that dominate system performance.
  - By **optimizing** these routines you can **reduce the system power consumption**
  - **Reduce the clock speed** needed for real-time operation.
- Optimization can turn an
  - infeasible system into a feasible one, or
  - an uncompetitive system into a competitive one.

Writing Assembly Code

- **For maximum performance**, you can optimize critical routines using **hand-written assembly**.
- **SO** we will address in this chapters three major Optimization Tools
  - ***Instruction scheduling****:* Reordering the instructions in a code sequence to avoid processor stalls.
  - ***Register allocation****:* Deciding how variables should be allocated to ARM registers or stack locations for maximum performance
  - ***Conditional execution****:* Accessing the full range of ARM condition codes and conditional instructions

Instruction Scheduling

- The time taken to execute instructions depends on the implementation pipeline.
  - (Assume **ARM9TDMI** pipeline timings)
- Instructions that are conditional on the value of the ARM condition codes in the *cpsr* take **one cycle** if the **condition is not met.**
- But if **met** then the following rules apply:

Instruction Scheduling

- **ALU** operations such as addition, subtraction, and logical operations take **one cycle**.
- **LOAD** instructions that load ***N* 32-bit** words of memory such as LDR and LDM take ***N* cycles** to issue
  - but the result of the last word loaded **is not available on the following one cycle**. updated load address is available on the next cycle
- **LOAD** instructions that load **16-bit** or **8-b**it data such as LDRB, LDRSB, LDRH, and LDRSH take **one cycle** to issue.
  - The load result is not available on the following **two cycles**.

Instruction Scheduling

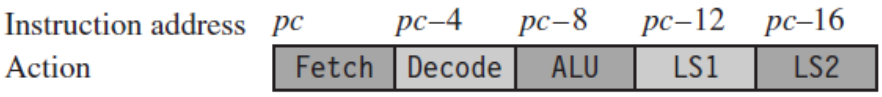
- **Branch** instructions take **three cycles**.
- **Store** instructions that store ***N* values** take ***N* cycles**.
  - An STM of a single value is exceptional, taking two cycles.
- **Multiply** instructions take a **varying** number of **cycles** depending on the value of the second operand in the product (**see Table D.6 in Section D.3**).

ARM9TDMI Processor Pipeline

- ***Fetch****:* Fetch from memory the instruction at address *pc*.
- ***Decode****:* Decode the instruction that was fetched in the previous cycle.
  - The processor also **reads the input operands from the register bank** if they are **not available via one of the forwarding paths**.
- ***ALU****:* Executes the instruction that was decoded in the previous cycle.
  - Note this instruction was originally fetched from address ***pc* − 8 (ARM state)** or *pc* − 4 (Thumb state).
  - Some instructions may spend several cycles in this stage.
  - Or do **addressing calculations**.

ARM9TDMI Processor Pipeline

- ***LS1:*** Load or store the data specified by a load or store instruction.
  - If the instruction is not a load or store, then this stage has no effect.
- ***LS2:*** Extract and zero- or sign-extend the data loaded by a byte or half word load instruction.
  - If the instruction is not a load of an 8-bit byte or 16-bit half word item, then this stage has no effect.



Example - 1

- This example shows the case where there is no interlock.

  ADD r0, r0, r1

  ADD r0, r0, r2
- This instruction pair takes **two cycles**. The ALU calculates *r0* + *r1* in one cycle
  - result is available for the ALU to calculate *r0* + *r2* in the second cycle.
- **Because it uses Forwarding found in ARM9TDMI.**

Example - 2

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	*ADD*	LDR	...
Cycle 3		...	ADD	—	LDR

- This example shows a **one-cycle interlock** caused by load use

  LDR r1, [r2, #4] 2 cycles

  ADD r0, r0, r1 1 cycle
- Takes **three cycles**.
- ALU calculates the address *r2* + 4 in the firstcycle while decoding the ADD instruction in parallel.
- Pipeline stalls for one cycle.
- Processor executes the ADD in the ALU on the third cycle

Example - 3

- This example shows a one-cycle interlock caused by delayed load use.

  LDRB r1, [r2, #1] 3 Cycles

  ADD r0, r0, r2 1 Cycle (within LDRB!)

  EOR r0, r0, r1 1 Cycle
- Takes **four cycles**.
- When EOR starts at cycle 3, LDRB need another cycle to write back r1.
- So ARM stalls EOR for 1 cycle more.

Scheduling of load instructions

- Load instructions occur frequently in compiled code, accounting for one third of all instructions!!
  - **Very important? Why?**
  - **Make it faster >>makes execution faster by 30%.**
- Careful scheduling of load instructions so that **pipeline stalls don't occur can** improve performance.
- The compiler attempts to schedule the code as best it can.
  - But still **needs user optimization**.

Scheduling of load instructions Example

- Let's consider an example of a memory-intensive task.
- The following function, *str_tolower*, copies a zero-terminated string of characters from *in* to *out*. It converts the string to lowercase in the process.
- **See next example in C and ARM assembly**

Scheduling of load instructions Example

```
void str_tolower(char *out, char *in)
{
  unsigned int c;

  do
  {
    c = *(in++);
    if (c>='A' && c<='Z')
    {
      c = c + ('a' -'A');
    }
    *(out++) = (char)c;
  } while (c);
}
```

Compiler Code

```
str_tolower
        LDRB    r2,[r1],#1    ; c = *(in++)
        SUB     r3,r2,#0x41   ; r3 = c -'A'
        CMP     r3,#0x19      ; if (c <='Z'-'A')
        ADDLS   r2,r2,#0x20   ;     c +='a'-'A'
        STRB    r2,[r0],#1    ; *(out++) = (char)c
        CMP     r2,#0         ; if (c!=0)
        BNE     str_tolower   ;     goto str_tolower
        MOV     pc,r14        ; return
```

Scheduling of load instructions Example

- Unfortunately, the **SUB instruction uses the value of c directly after the LDRB** instruction that loads c. Consequently, the ARM9TDMI pipeline will **stall for two cycles**.
- Everything following the load of c depends on its value!
- **But**
  - **two ways you can alter the structure of the algorithm to avoid the cycles by using assembly**
- This takes **11 cycles** to execute. See how 11??

Load Scheduling by Preloading

```
out     RN 0    ; pointer to output string
in      RN 1    ; pointer to input string

c       RN 2    ; character loaded
t       RN 3    ; scratch register
        ; void str_tolower_preload(char *out, char *in)
        str_tolower_preload
        LDRB    c, [in], #1       ; c = *(in++)
loop
        SUB     t, c, #'A'        ; t = c-'A'
        CMP     t, #'Z'-'A'       ; if (t <= 'Z'-'A')
        ADDLS   c, c, #'a'-'A'    ;   c += 'a'-'A';
        STRB    c, [out], #1      ; *(out++) = (char)c;
        TEQ     c, #0             ; test if c==0
        LDRNEB  c, [in], #1       ; if (c!=0) { c=*in++;
        BNE     loop              ;             goto loop; }
        MOV     pc, lr            ; return
```

- we load the data required for the loop at the end of the previous loop.
- To get performance improvement with little increase in code size.
- one instruction longer than the C version, but we save two cycles for each inner loop iteration.
- reduces the loop from **11 cycles** per character to **9 cycles** per character on an ARM9TDMI, giving a **1.22 times** speed improvement

Load Scheduling by Unrolling

- Works by unrolling and then interleaving the body of the loop.
  - For Example: loop iterations *i*, *i* + 1, *i* + 2 interleaved.
- When the result of an operation from loop *i* is not ready, we can perform an operation from loop *i* + 1.

Load Scheduling by Unrolling Example

- The next program applies **load scheduling** by **unrolling** to the *str_tolower* function.
- **See**
  - Very efficient
  - Requires **7 cycles** per character on ARM9TDMI.
  - **1.57 times** speed increase over the original str_tolower.
  - We use **conditional instructions** to avoid storing characters that are past the end of the string.
  - Improvements cost: The routine is more than **double** the code size of the original implementation
  - For **time-critical** parts of an application where you know the data size is large

```
out     RN 0    ; pointer to output string
in      RN 1    ; pointer to input string
ca0     RN 2    ; character 0
t       RN 3    ; scratch register

ca1     RN 12   ; character 1
ca2     RN 14   ; character 2
        ; void str_tolower_unrolled(char *out, char *in)
        str_tolower_unrolled
        STMFD   sp!, {lr}         ; function entry
loop_next3
        LDRB    ca0, [in], #1     ; ca0 = *in++;
        LDRB    ca1, [in], #1     ; ca1 = *in++;
        LDRB    ca2, [in], #1     ; ca2 = *in++;
        SUB     t, ca0, #'A'      ; convert ca0 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca0, ca0, #'a'-'A'
        SUB     t, ca1, #'A'      ; convert ca1 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca1, ca1, #'a'-'A'
        SUB     t, ca2, #'A'      ; convert ca2 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca2, ca2, #'a'-'A'
        STRB    ca0, [out], #1    ; *out++ = ca0;
        TEQ     ca0, #0           ; if (ca0!=0)
        STRNEB  ca1, [out], #1    ;   *out++ = ca1;
        TEQNE   ca1, #0           ; if (ca0!=0 && ca1!=0)
        STRNEB  ca2, [out], #1    ;   *out++ = ca2;
        TEQNE   ca2, #0           ; if (ca0!=0 && ca1!=0 && ca2!=0)
        BNE     loop_next3        ;   goto loop_next3;
        LDMFD   sp!, {pc}         ; return;
```

Register Allocation

- You can use **14** of the 16 visible ARM registers to hold **general-purpose data**.
- For a function to be AAPCS compliant it must preserve the callee values of registers ***r4* to *r11***.
- Use the following template for optimized assembly routines requiring many registers:

Register Allocation

```
routine_name
        STMFD sp!,       {r4-r12, lr}      ; stack saved registers
          ; body of routine
          ; the fourteen registers r0-r12 and lr are available
        LDMFD sp!,       {r4-r12, pc}      ; restore registers and return

routine_name
       STMFD sp!,      {r4-r12, lr}      ; stack saved registers
         ; body of routine
         ; registers r0-r12 and lr available
       LDMFD sp!,      {r4-r12, lr}      ; restore registers
       BX         lr                     ; return, with mode switch
```