

COL216

Computer Architecture

Assembly Language
Programming
10th Jan, 2022

Machine Language

- Primitive compared to HLLs
- Language understood by Machine
- Easily interpreted by hardware
- Programmer's view of hardware

Instruction set design goals

- Maximize performance
- Minimize cost, energy consumption
- Reduce design time

ARM instructions so far

- add, sub
- mov
- cmp
- mul
- ldr, str
- ldrb, strb, ldrh, strh
- b, beq, bne, blt, ble, bgt, bge

Procedural Abstraction



What is required?

- Control flow (call and return)
- Data flow (parameter passing)
- Local and global storage allocation
- Take care of nesting
- Take care of recursion

Control flow - call

```
...  
X: func ( );  
...  
...  
Y: func ( );  
...
```

```
...  
X: bl func  
...  
...  
Y: bl func  
...
```

Control flow - return

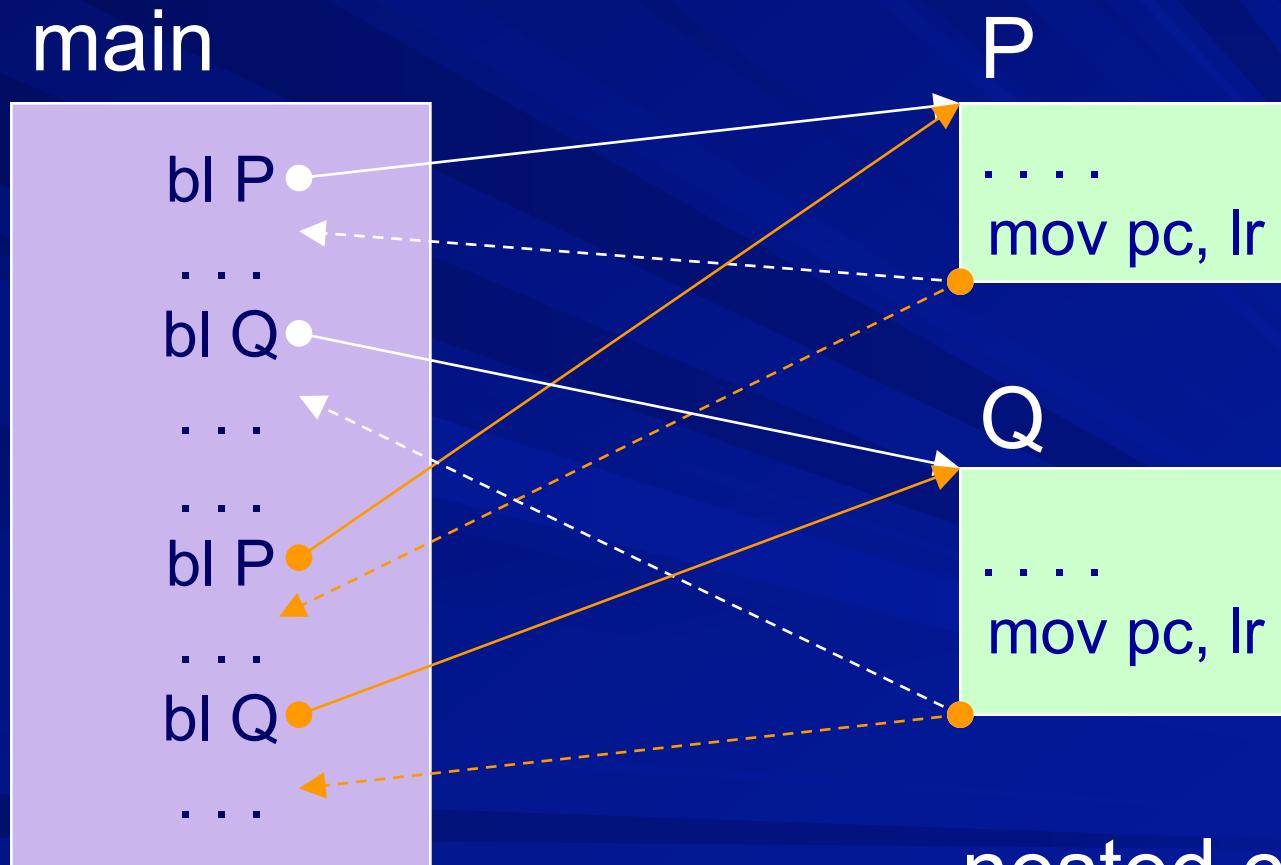
```
void func ( ) {  
    ...  
    ...  
    return;  
}
```

```
func:  
    ...  
    ...  
    mov pc, lr
```

Registers with special role

- r15 pc (program counter)
- r14 lr (link register)
- r13 sp (stack pointer)

Call and Return



nested calls?
recursive calls?

Passing parameters

through registers

caller :

....

....

move parameters
into registers

bl callee

take result from register

....

....

callee :

....

....

access parameters
in registers

....

....

....

mov pc, lr

Passing parameters

through stack

caller :

.....

.....

push parameters
into stack

bl callee

pop results from stack

.....

.....

callee :

.....

.....

access parameters
in stack

.....

.....

.....

mov pc, lr

Conventions

- First 4 parameters through r0, r1, r2, r3
- Result in r0
- Beyond this, use stack
- Callee can destroy r0,r1,r2,r3,r12
- It should preserve other registers, except pc
- Caller should preserve r0,r1,r2,r3,r12

Saving and restoring registers

caller :

.....

.....

save registers

move parameters
into registers

bl callee

take result from register
restore registers

.....

.....

callee :

save registers

.....

.....

access parameters
in registers

.....

.....

.....

restore registers

mov pc, lr

Example with procedure/function

- GCD of n numbers

$G = A[0]$

for ($i = 1; i < n; i++$)

$G = \text{gcd}(G, A[i])$

- Compare with sum of n numbers

$S = A[0]$

for ($i = 1; i < n; i++$)

$S = S + A[i]$

Array sum program

```
.equ SWI_Exit 0x11
.text
    mov r1, #0
    ldr r3, =AA
    add r6, r3, #400
L:   ldr r5, [r3, #0]
    add r1, r1, r5
    add r3, r3, #4
    cmp r3, r6      @ r6 = q
    blt L
    swi SWI_Exit
.data
AA: .space 400
.end
```

Sum => GCD

```
.equ SWI_Exit. 0x11
.text
L:    mov  r1, #0
      ldr  r3, =AA
      add  r6, r3, #400
      ldr  r5, [r3, #0]
      add  r1, r1, r5
      add  r3, r3, #4
      cmp  r3, r6      @ r6 = q
      blt  L
      swi  SWI_Exit
.data
AA:   .space 400
.end
```

The diagram illustrates the assembly code flow. It shows a sequence of instructions starting with `mov r1, #0`, followed by `ldr r3, =AA`, `add r6, r3, #400`, and `ldr r1, [r3, #0]`. These four instructions are highlighted in a rounded rectangle and have an arrow pointing to a second rounded rectangle containing `ldr r3, =AA`, `add r6, r3, #400`, `ldr r1, [r3, #0]`, and `add r3, r3, #4`. Below this, another arrow points from the `add r3, r3, #4` instruction to a third rounded rectangle containing the expression `r1 = gcd (r1, r5)`.

Array GCD program

```
ldr r3, =AA  
add r6, r3, #400  
ldr r1, [r3, #0]  
add r3, r3, #4  
L: ldr r5, [r3, #0]  
    r1 = gcd (r1, r5) → r0 = gcd (r0, r1)  
    add r3, r3, #4  
    cmp r3, r6      @ r6 = q  
    blt L
```

follow conventions

Array GCD program

```
ldr r3, =AA  
add r6, r3, #400  
ldr r0, [r3, #0]  
add r3, r3, #4  
L: ldr r1, [r3, #0]  
r0 = gcd (r0, r1)  
add r3, r3, #4  
cmp r3, r6      @ r6 = q  
blt L
```

follow conventions

r3 => r4

Array GCD program

```
ldr r4, =AA
add r6, r4, #400
ldr r0, [r4, #0]
add r4, r4, #4
L: ldr r1, [r4, #0]
    r0 = gcd (r0, r1)
    add r4, r4, #4
    cmp r4, r6      @ r6 = q
    blt L
```

Array GCD program with “bl”

```
ldr r4, =AA
add r6, r4, #400
ldr r0, [r4, #0]
add r4, r4, #4
L: ldr r1, [r4, #0]
    bl gcd
    add r4, r4, #4
    cmp r4, r6      @ r6 = q
    blt L
```

GCD function

```
gcd:    cmp r0, r1
        beq ret
        blt sub10
sub01: sub r0, r0, r1
        b    gcd
sub10: sub r1, r1, r0
        b    gcd
ret:   mov pc, lr
```

DP (data processing) instructions

- | | |
|--------------|--------------------------|
| ■ Arithmetic | operation dest, op1, op2 |
| ■ Logical | operation dest, op1, op2 |
| ■ Move | operation dest, src |
| ■ Compare | operation op1, op2 |
| ■ Test | operation op1, op2 |

DP instructions: Arithmetic

- add
- sub
- rsb
- adc
- sbc
- rsc



reverse subtract: $op2 - op1$

add / sub / rsb with carry

DP instructions: Logical

- and bit by bit logical AND
- orr bit by bit logical OR
- eor bit by bit logical XOR
- bic bit clear: op1 and not op2

DP instructions: Move

- mov dest <= src
 - mvn dest <= not src

DP instructions: Compare

- cmp $op1 - op2$
- cmn $op1 - (-op2)$

DP instructions: Test

- **tst** op1 and op2
- **teq** op1 eor op2

DT (data transfer) instructions

- ldr / str load / store word
- ldrb / strb load / store byte
- ldrh / strh load / store half word
- ldrsb / ldrsh load signed byte / half word
(sign extension to fill the register)
- ldm / stm load / store multiple
(any subset of registers can be specified)

Comparison in ARM

- Signed comparison:

equal	beq
not equal	bne
greater or equal	bge
less than	blt
greater than	bgt
less or equal	ble

- Unsigned comparison

equal	beq
not equal	bne
higher or same	bhs
lower	blo
higher	bhi
lower or same	bls

Status flags

These are part of Program Status Register

- N Negative
- Z Zero
- C Carry
- V Overflow

Condition codes and flags

0	eq	$Z = 1$
1	ne	$Z = 0$
2	hs / cs (C set)	$C = 1$
3	lo / cc (C clear)	$C = 0$
4	mi (minus)	$N = 1$
5	pl (plus)	$N = 0$
6	vs (V set)	$V = 1$
7	vc (V clear)	$V = 0$

8	hi	$C = 1$ and $Z = 0$
9	ls	$C = 0$ or $Z = 1$
10	ge	$N = V$
11	lt	$N \neq V$
12	gt	$N = V$ and $Z = 0$
13	le	$N \neq V$ or $Z = 1$
14	al	flags ignored

Assembler directives

.text

.data

.end

.space

.word

.byte

.ascii

.asciz

.equ

Input Output in ARM

SWI instruction

- Instruction to invoke some service provided by system software

Use of SWI in ARMSim

- input/output from/to stdin/stdout
- input/output from/to files
- opening/closing of files
- Halt execution
- ...

I/O example in ARMsim# 1.91

```
ldr r0, =message  
swi 0x02      @ write on stdout
```

....

message: .asciz “Welcome\n”

I/O example in ARMsim# 2.01

```
ldr    r1, =param
mov    r4, #1          @ file #1 is stdout
str    r4, [r1]
ldr    r4, =message
str    r4, [r1, #4]
mov    r4, #8          @ number of bytes
str    r4,[r1,#8]
mov    r0, #5          @ code for write
swi    0x123456
...

```

param: .word 0, 0, 0

message: .ascii "Welcome\n"

Software Interrupts

- Similar terms –
 - System calls, Traps, Exceptions
- Are there hardware interrupts?
- Hardware interrupts are caused by events/conditions detected by hardware
 - intentional : e.g., I/O event
 - unintentional : e.g., hardware fault, power outage, arithmetic overflow
- Software interrupts are caused by specific instructions

Response to interrupts

- Response mechanism in both cases (h/w and s/w interrupts) is same
- Execution of some code
 - interrupt handler or interrupt service routine (ISR)

ISR vs normal subroutine

- Processors have two or more modes
 - normal mode / user mode
 - privileged mode / kernel mode / supervisor mode
- Application program executes in user mode
- To do certain privileged tasks, execution of some kernel code is required
- ISR executes in privileged mode, provides controlled access to kernel functions

Thank you