

Programming the ARM Processor

Mark McDermott

Agenda

- **Assembly Language Programming**
- **C Programming**

GNU compiler and binutils

- **TLL6219 GNU compiler and binutils**

- gcc: GNU C compiler
- as: GNU assembler
- ld: GNU linker
- gdb: GNU project debugger

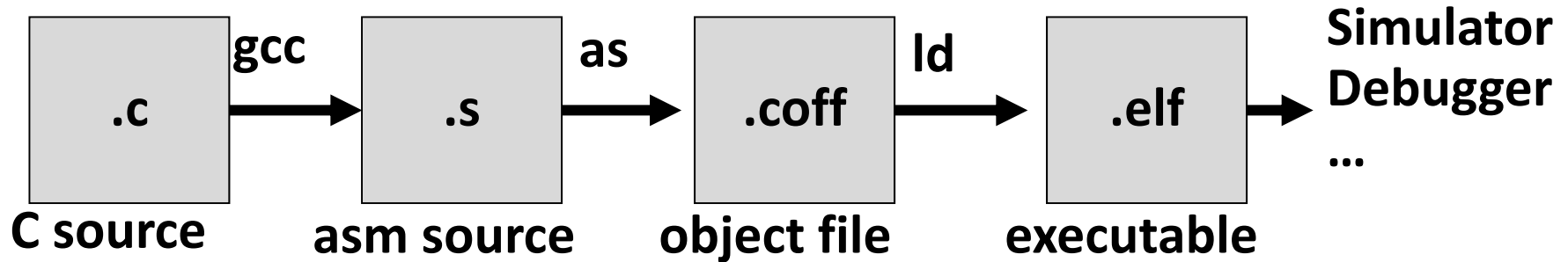
- **There are two types of compilers**

- One is for code that is compiled to run on top of Linux
- The other is for “Bare Metal” code.

```
# -----
# Use this configuration if you want to compile code for Linux operation
# -----
#PATH=$PATH:/usr/local/packages/arm/arm-2008q1/bin
#export PATH
# -----
# Use this configuration if you want to do Bare-Metal ARM code (EABI)
# -----
#PATH=$PATH:/home/ecelrc/faculty/mcdermot/CodeSourcery/Sourcery_G++_Lite/bin
#export PATH
```

Pipeline

- **COFF (common object file format)**
- **ELF (extended linker format)**
- **Segments in the object file**
 - **Text: code**
 - **Data: initialized global variables**
 - **BSS: uninitialized global variables**



Gnu AS program format

```
.file "test.s"
.text
.global main
.type main, %function
```

main:

```
MOV R0, #100
ADD R0, R0, R0
SWI #11
.end
```

Gnu AS program format

```
.file "test.s"
```

```
.text
```

export variable → `.global main`

```
.type main, %function
```

```
main:
```

```
MOV R0, #100
```

```
ADD R0, R0, R0
```

```
SWI #11
```

signals the end of the program → `.end`

```
.end
```

↑
 set the type of a symbol to be either a function or an object

↙ call interrupt to end the program

ARM assembly program

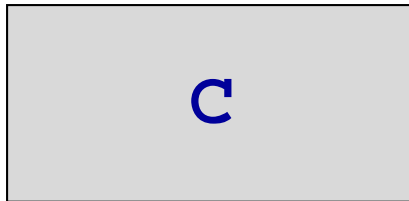
label	operation	operand	comments
main:	LDR	R1, value	// load value
	STR	R1, result	
	SWI	#11	
value:	.word	0x0000C123	
result:	.word	0	

Control structures

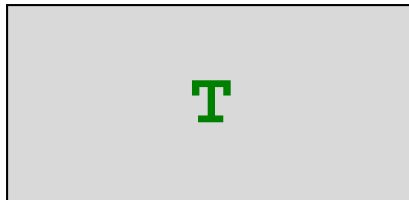
- **Flow of control:**
 - Sequence.
 - Decision: if-then-else, switch
 - Iteration: repeat-until, do-while, for

if statements

if **C** then **T** else **E**

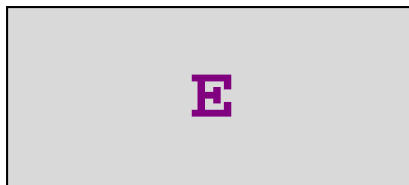


BNE else



B endif

else:

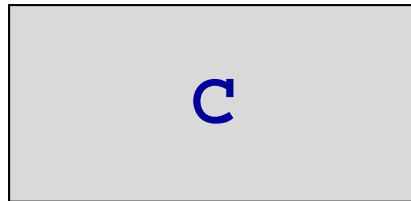


endif:

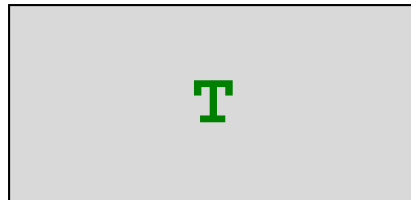
```
// find maximum
if (R0>R1) then R2:=R0
else R2:=R1
```

if statements

if **C** then **T** else **E**



BNE else



B endif

else:



endif:

```
// find maximum
if (R0>R1) then R2:=R0
else R2:=R1
```

```
CMP R0, R1
```

```
BLE else
```

```
MOV R2, R0
```

```
B endif
```

```
else: MOV R2, R1
```

```
endif:
```

if statements

Two other options:

```
CMP    R0, R1
```

```
MOVGT  R2, R0
```

```
MOVLE  R2, R1
```

```
MOV    R2, R0
```

```
CMP    R0, R1
```

```
MOVLE  R2, R1
```

```
// find maximum
if (R0>R1) then R2:=R0
else R2:=R1
```

```
CMP R0, R1
```

```
BLE else
```

```
MOV R2, R0
```

```
B endif
```

```
else: MOV R2, R1
```

```
endif:
```

if statements

```
if (R1==1 || R1==5 || R1==12) R0=1;
```

```
TEQ    R1, #1      ...
TEQNE  R1, #5      ...
TEQNE  R1, #12     ...
MOVEQ  R0, #1      BNE fail
```

if statements

```
if (R1==0) zero
else if (R1>0) plus
else if (R1<0) neg
```

```
TEQ    R1, #0
BMI    neg
BEQ    zero
BPL    plus
```

```
neg:   ...
       B exit
Zero:  ...
       B exit
       ...
```

Multi-way branches

```

CMP R0, #`0'
BCC other    // less than `0'
CMP R0, #`9'
BLS digit   // between `0' and `9'
-----
CMP R0, #`A'
BCC other
CMP R0, #`Z'
BLS letter  // between `A' and `Z'
-----
CMP R0, #`a'
BCC other
CMP R0, #`z'
BHI other   // not between `a' and `z'
-----

```

letter: ...

Switch statements

```
switch (exp) {
    case c1: s1; break;
    case c2: s2; break;
    ...
    case cN: sN; break;
    default: sD;
}
```

```
e=exp;
if (e==c1) {s1}
else
    if (e==c2) {s2}
    else
        ...
```

Switch statements

```
switch (R0) {
    case 0: S0; break;
    case 1: S1; break;
    case 2: S2; break;
    case 3: S3; break;
    default: err;
}
```

```
CMP R0, #0
BEQ S0
CMP R0, #1
BEQ S1
CMP R0, #2
BEQ S2
CMP R0, #3
BEQ S3
```

The range is between 0 and N

Slow if N is large

```
err: ...
    B exit
S0: ...
    B exit
```

Switch statements

ADR R1, JMPTBL

CMP R0, #3

LDRLS PC, [R1, R0, LSL #2]

err:...

B exit

S0: ...

JMPTBL:

.word S0

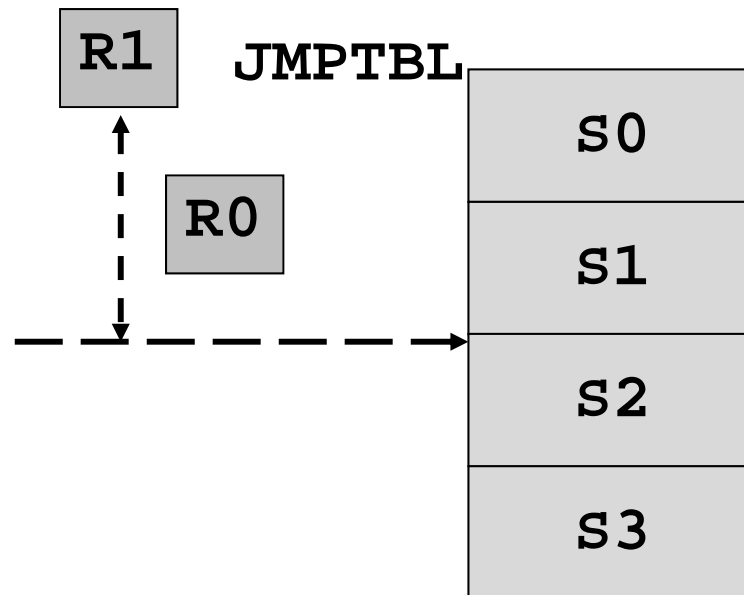
.word S1

.word S2

.word S3

What if the range is between M and N?

For larger N and sparse values, we could use a hash function.



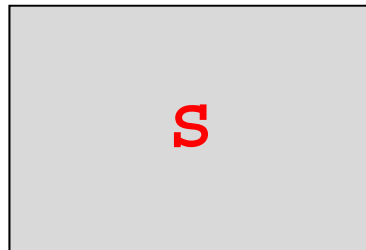
Iteration

- **repeat-until**
- **do-while**
- **for**

repeat loops

do { **S** } while (**C**)

loop:

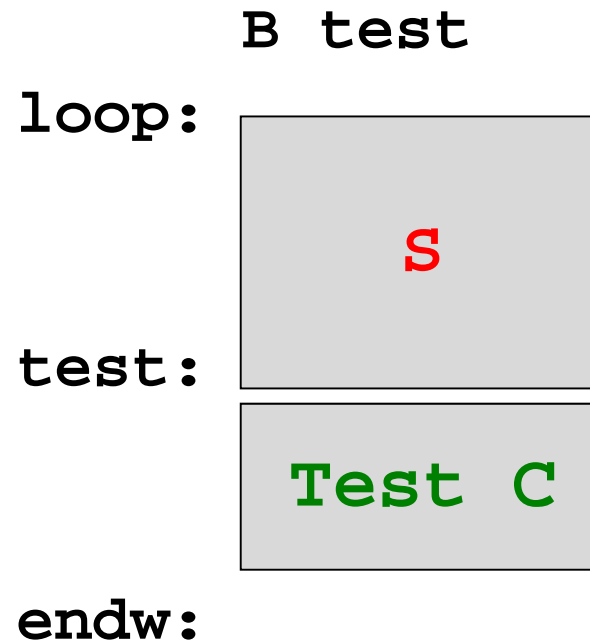
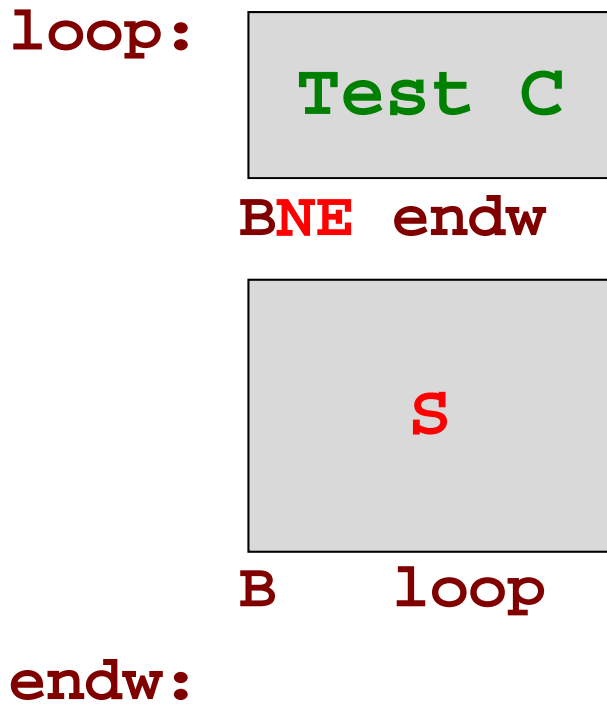


BEQ loop

endw:

while loops

```
while ( C ) { S }
```



while loops

while (**C**) { **S** }

B test

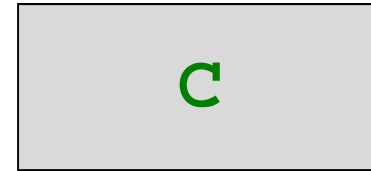
loop:



test:



endw:

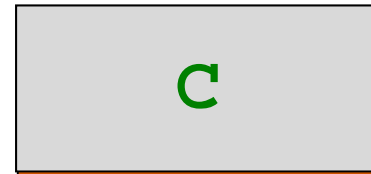


BNE endw

loop:



test:



endw:

GCD

```
int gcd (int i, int j)
{
    while (i!=j)
    {
        if (i>j)
            i -= j;
        else
            j -= i;
    }
}
```

GCD

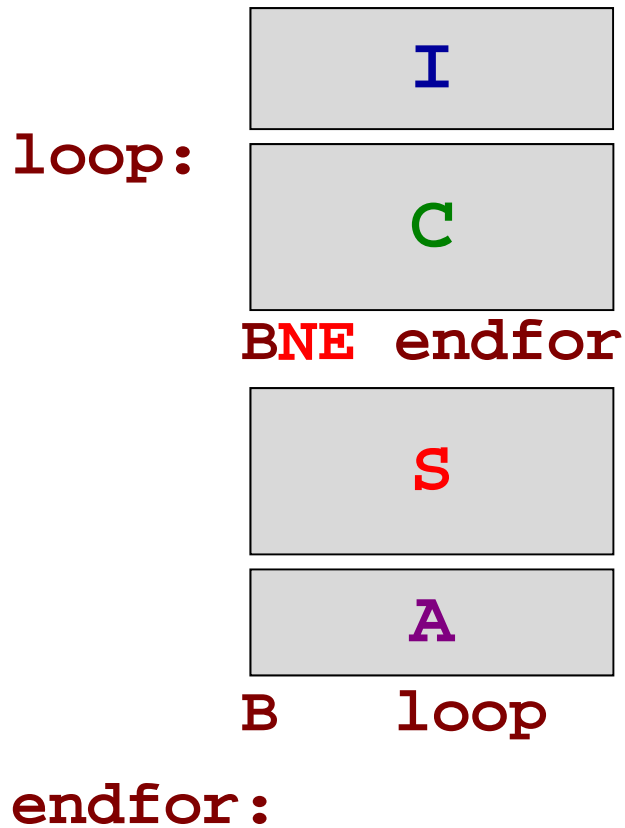
```

Loop:  CMP    R1, R2
        SUBGT  R1, R1, R2
        SUBLT  R2, R2, R1
        BNE   loop
    
```

for loops

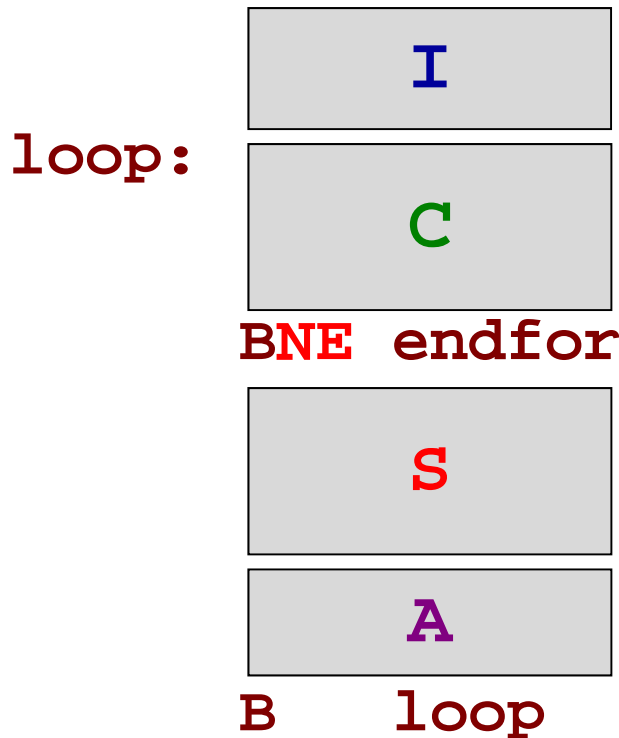
```
for ( I ; C ; A ) { S }
```

```
for (i=0; i<10; i++)  
{ a[i]:=0; }
```



for loops

for (**I** ; **C** ; **A**) { **S** }



endfor:

```
for (i=0; i<10; i++)
    { a[i]:=0; }
```

```
MOV R0, #0
ADR R2, A
MOV R1, #0
loop: CMP R1, #10
      BGE endfor
      STR R0, [R2, R1, LSL #2]
      ADD R1, R1, #1
      B loop
endfor:
```

for loops

```
for (i=0; i<10; i++)
    { do something; }
```

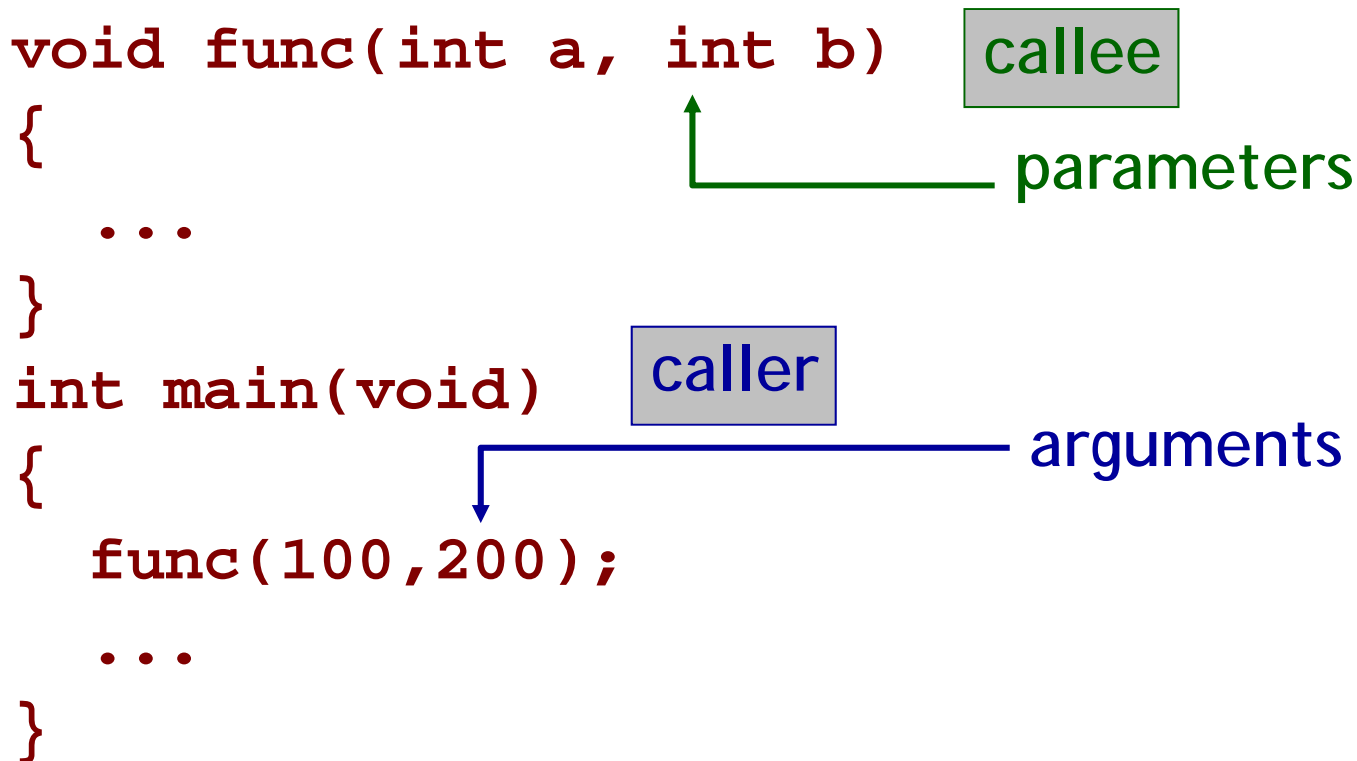
Execute a loop for a constant of times.

```
        MOV R1, #0
loop:   CMP R1, #10
        BGE endfor
        // do something
        ADD R1, R1, #1
        B    loop
endfor:
```

```
        MOV R1, #10
loop:
        // do something
        SUBS R1, R1, #1
        BNE loop
endfor:
```

Procedures

- Arguments: expressions passed into a function
- Parameters: values received by the function
- Caller and callee



Procedures

main:

```

    ...
BL func
    ...

.end

```

func:

```

    ...
    ...


.end

```

- **How to pass arguments? By registers? By stack? By memory?
In what order?**

Procedures

```

main: caller
    // use R5
    BL func 
    // use R5
    ...
    ...
    .end

callee
func:
    ...
    // use R5
    ...
    ...
    .end
    
```

- How to pass arguments? By registers? By stack? By memory?
In what order?
- Who should save R5? Caller? Callee?

Procedures (caller save)

<pre>main: caller // use R5 // save R5 BL func // restore R5 // use R5 .end</pre>		<pre>func: callee ... // use R5 .end</pre>
---	--	--

- How to pass arguments? By registers? By stack? By memory?
In what order?
- Who should save R5? Caller? Callee?

Procedures (callee save)

<pre>main: caller // use R5 BL func // use R5 .end</pre>		<pre>func: callee // save R5 ... // use R5 //restore R5 .end</pre>
---	--	---

- How to pass arguments? By registers? By stack? By memory?
In what order?
- Who should save R5? Caller? Callee?

ARM Procedure Call Standard (APCS)

- **ARM defines a set of rules for procedure entry and exit so that**
 - Object codes generated by different compilers can be linked together
 - Procedures can be called between high-level languages and assembly

- **APCS defines**
 - Use of registers
 - Use of stack
 - Format of stack-based data structure
 - Mechanism for argument passing

APCS register usage convention

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

APCS register usage convention

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

- **Used to pass the first 4 parameters**
- **Caller-saved if necessary**

APCS register usage convention

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

- **Register variables, must return unchanged**
- **Callee-saved**

APCS register usage convention

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

- **Registers for special purposes**
- **Could be used as temporary variables if saved properly.**

Argument passing

- The first four word arguments are passed through R0 to R3.
- Remaining parameters are pushed into stack in the reverse order.
- Procedures with less than four parameters are more effective.

Return value

- One word value in R0
- A value of length 2~4 words (R0-R1, R0-R2, R0-R3)

Function entry/exit

- A simple leaf function with less than four parameters has the minimal overhead. 50% of calls are to leaf functions

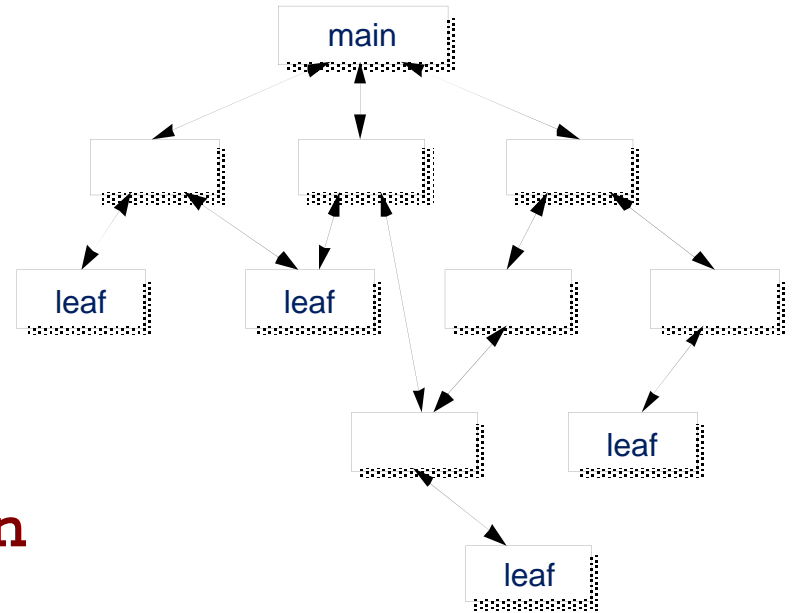
BL leaf1

...

leaf1: ...

...

MOV PC, LR // return



Function entry/exit

- Save a minimal set of temporary variables

```
BL leaf2
```

```
...
```

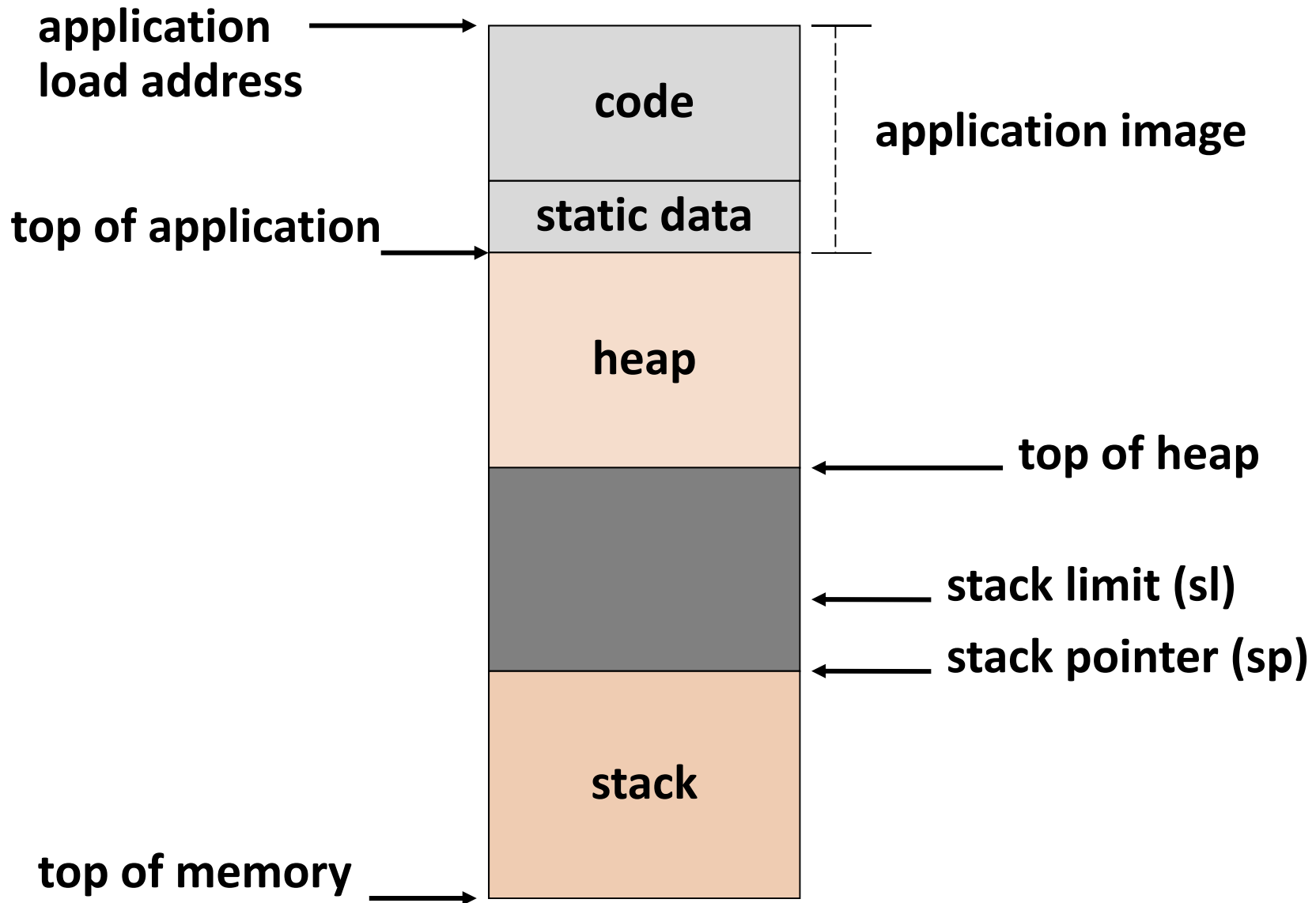
```
leaf2: STMFD sp!, {regs, lr} // save
```

```
...
```

```
LDMFD sp!, {regs, pc} // restore and  

// return
```

Standard ARM C program address space



Accessing operands

- **A procedure often accesses operand in the following ways**
 - An argument passed on a register: no further work
 - An argument passed on the stack: use stack pointer (R13) relative addressing with an immediate offset known at compiling time
 - A constant: PC-relative addressing, offset known at compiling time
 - A local variable: allocate on the stack and access through stack pointer relative addressing
 - A global variable: allocated in the static area and can be accessed by the static base relative (R9) addressing

Procedure

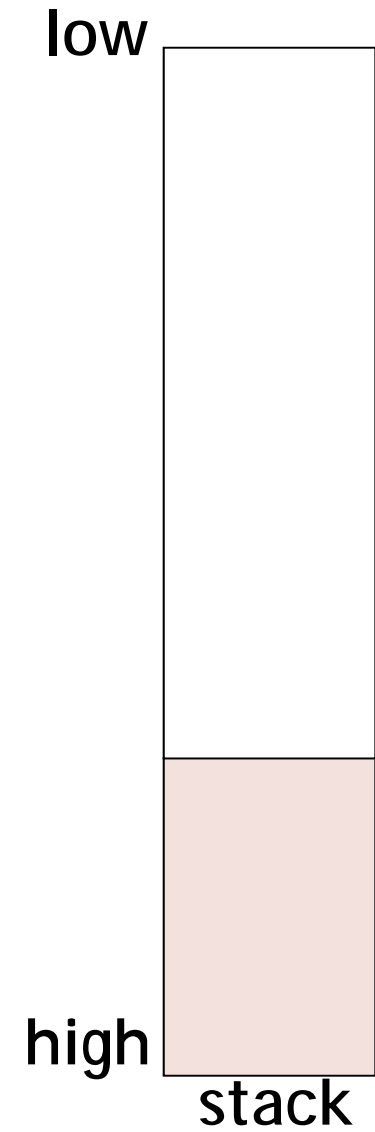
main:

LDR R0, #0

...

BL func

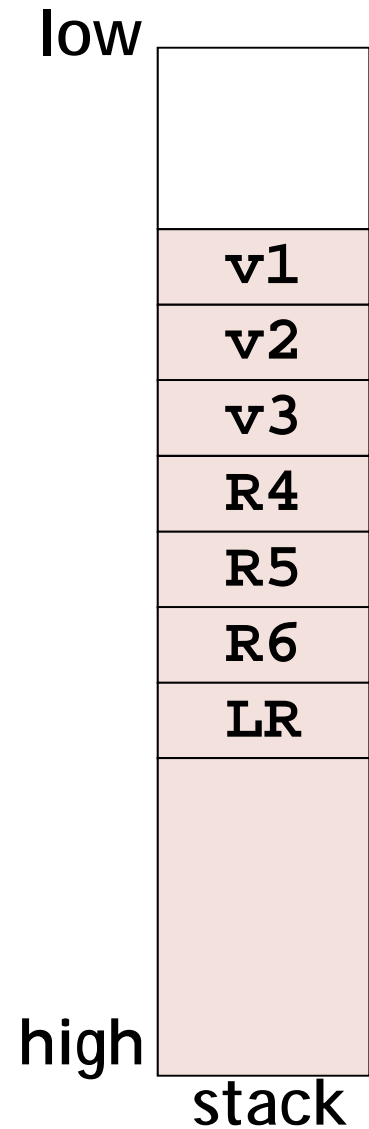
...



Procedure

```

func:    STMFD SP!, {R4-R6, LR}
           SUB   SP, SP, #0xC
           ...
           STR   R0, [SP, #0] // v1=a1
           ...
           ADD   SP, SP, #0xC
           LDMFD SP!, {R4-R6, PC}
    
```



Block copy example

```
void bcopy(char *to, char *from, int n)
{
    while (n-- > 0)
        *to++ = *from++;
}
```

Block copy example

```
// arguments: R0: to, R1: from, R2: n
bcopy:   TEQ    R2, #0
         BEQ    end
loop:    SUB    R2, R2, #1
         LDRB  R3, [R1], #1
         STRB  R3, [R0], #1
         B     bcopy
end:     MOV    PC, LR
```

Block copy example

```
// arguments: R0: to, R1: from, R2: n
```

```
// rewrite "n--" as "--n>=0"
```

```
bcopy:  SUBS    R2, R2, #1
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        BPL    bcopy
        MOV    PC, LR
```

Block copy example

```
// arguments: R0: to, R1: from, R2: n
// assume n is a multiple of 4; loop
// unrolling
```

```
bcopy:  SUBS    R2, R2, #4
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        BPL    bcopy
        MOV    PC, LR
```

Block copy example

```
// arguments: R0: to, R1: from, R2: n
// n is a multiple of 16;
bcopy:  SUBS    R2, R2, #16
        LDRPL  R3, [R1], #4
        STRPL  R3, [R0], #4
        LDRPL  R3, [R1], #4
        STRPL  R3, [R0], #4
        LDRPL  R3, [R1], #4
        STRPL  R3, [R0], #4
        LDRPL  R3, [R1], #4
        STRPL  R3, [R0], #4
        BPL    bcopy
        MOV    PC, LR
```

Block copy example

```
// arguments: R0: to, R1: from, R2: n
// n is a multiple of 16;
bcopy:  SUBS    R2, R2, #16
        LDMPPL R1!, {R3-R6}
        STMPPL R0!, {R3-R6}
        BPL    bcopy
        MOV    PC, LR

// could be extend to copy 40 byte at a time
// if not multiple of 40, add a copy_rest
loop
```

Search example

```

int main(void)
{
    int a[10]={7,6,4,5,5,1,3,2,9,8};
    int i;
    int s=4;

    for (i=0; i<10; i++)
        if (s==a[i]) break;
    if (i>=10) return -1;
    else return i;
}
  
```

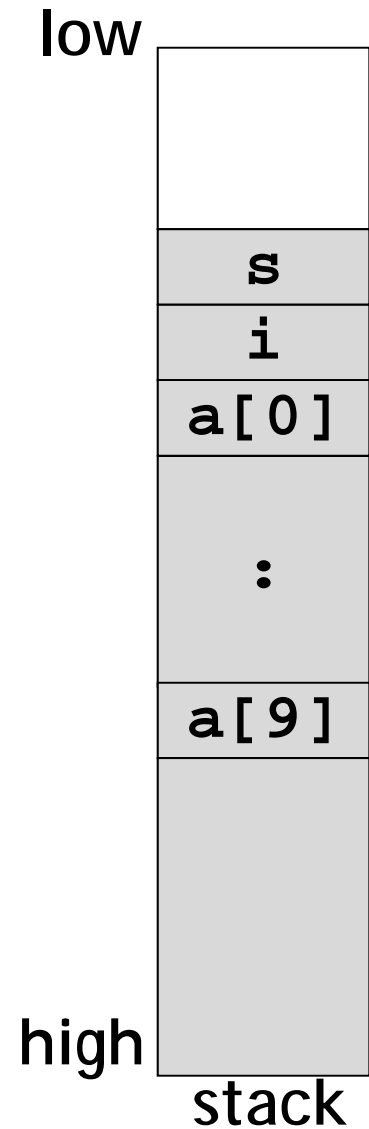
Search

```

        .section      .rodata
.LC0:
        .word        7
        .word        6
        .word        4
        .word        5
        .word        5
        .word        1
        .word        3
        .word        2
        .word        9
        .word        8
    
```

Search

```
.text
.global      main
.type        main, %function
main: sub     sp, sp, #48
      adr     r4, L9          // =.LC0
      add     r5, sp, #8
      ldmia  r4!, {r0, r1, r2, r3}
      stmia  r5!, {r0, r1, r2, r3}
      ldmia  r4!, {r0, r1, r2, r3}
      stmia  r5!, {r0, r1, r2, r3}
      ldmia  r4!, {r0, r1}
      stmia  r5!, {r0, r1}
```



Optimization

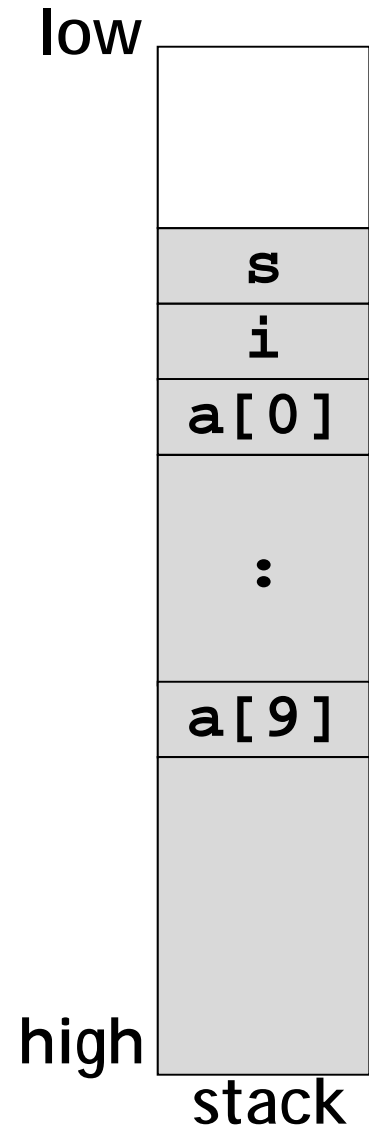
- **Remove unnecessary load/store**
- **Remove loop invariant**
- **Use addressing mode**
- **Use conditional execution**

Search (remove load/store)

```

mov  r1, #4
str  r3, [sp, #0] // s=4
mov  r0, #0
str  r3, [sp, #4] // i=0

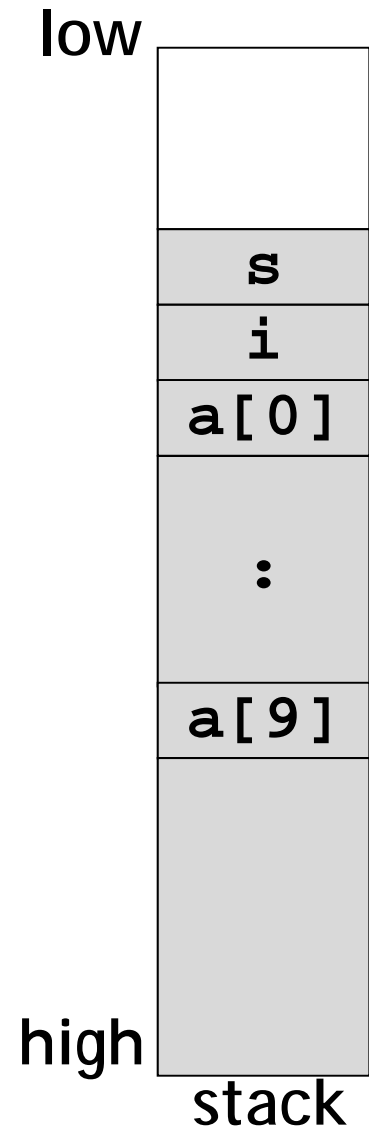
loop: ldr  r0, [sp, #4] // r0=i
      cmp  r0, #10 // i<10?
      bge  end
ldr  r1, [sp, #0] // r1=s
mov  r2, #4
mul  r3, r0, r2
add  r3, r3, #8
ldr  r4, [sp, r3] // r4=a[i]
    
```



Search (loop invariant/addressing mode)

```

mov r1, #4
str r3, [sp, #0] // s=4
mov r0, #0
str r3, [sp, #4] // i=0
mov r2, sp, #8
loop: ldr r0, [sp, #4] // r0=i
      cmp r0, #10 // i<10?
      bge end
ldr r1, [sp, #0] // r1=s
mov r2, #4
mul r3, r0, r2
add r3, r3, #8
      ldr r4, [r2, r0, LSL #2] / r4=a[i]
    
```



Optimization Summary

- Remove unnecessary load/store
- Remove loop invariant
- Use addressing mode
- Use conditional execution

- From 22 words to 13 words and execution time is greatly reduced.

Agenda

- **Assembly Language Programming**
- **C Programming**

Agenda

- **Assembly Language Programming**
- **C Programming**

C Programming in Embedded Systems

- **Assembly language**
 - dependent of processor architecture
 - cache control, registers, program status, interrupt
- **High-level language**
 - memory model
 - independent of processor architecture (partially true)
- **Advantages and disadvantages**
 - performance
 - code size
 - software development and life cycle

Manage IO Operations Using C

- Access memory-mapped IO – pointers
- Example

```
#define MX_REG_READ(a, val) ((val) = *(volatile UNIT32 *)(a))
#define MX_REG_WRITE(a, val) (*(volatile UNIT32 *)(a) = (val))
```

```
#define UART_CR          0x90003800
#define UART_SR          0x90003400
#define UART_RX_INT_EN  (1<<4)
#define UART_TX_INT_EN  (1<<8)
```

```
UNIT32 CR_word=0;
```

```
CR_word |= UART_RX_INT_EN | UART_TX_INT_EN;
MX_REG_WRITE(UART_CR, CR_word);
```

Bit Manipulation

■ Boolean operation

- operate on 1 (true) and 0 (false)
- `(2 || !6) && 7` ??

■ Bitwise operation

- operate on individual bit positions within the operands
- `(2 | ~6) & 7 = (0x0002 OR 0xFFF1) AND 0x0007`

if (bits & 0x0040)

bits |= (1 <<7)

long integer: bits &= ~(1L << 7)

Operation	Boolean op.	Bitwise op.
AND	<code>&&</code>	<code>&</code>
OR	<code> </code>	<code> </code>
XOR	unsupported	<code>^</code>
NOT	<code>!</code>	<code>~</code>

if (bits & (1 <<6))

bits &= ~(1 <<7)

Bit Fields

typedef struct

```
{
WORD16    x      :7,
          y      :6,
          z      :3;
} IO_WORD
```



- Bit fields: signed or unsigned integer (char)
- Can be referenced as regular structure

```
if IO_WORD    GIO, then GIO.x GIO.y and GIO.z are valid
if WORD16    GIO,
    y = (GIO >>3 ) & 0x003F;
    GIO |= (x &0x007F) << 9;
```

Variant Access

- **An object with a variety of organizations – provides different views**

typedef union

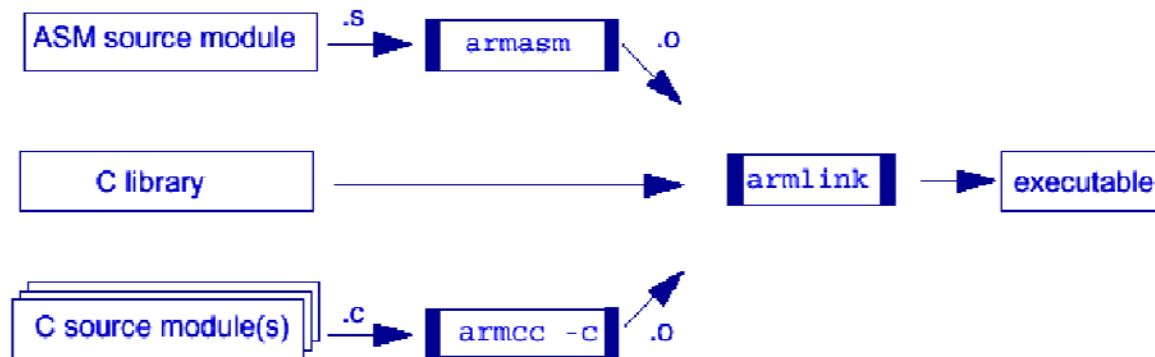
```
{
    unsigned int xyz;
    IO_WORD    iow;
} PORT_DEF
PORT_DEF     port;
```

port.xyz is an integer

port.iow.x, port.iow.y

Interface C and Assembly Language

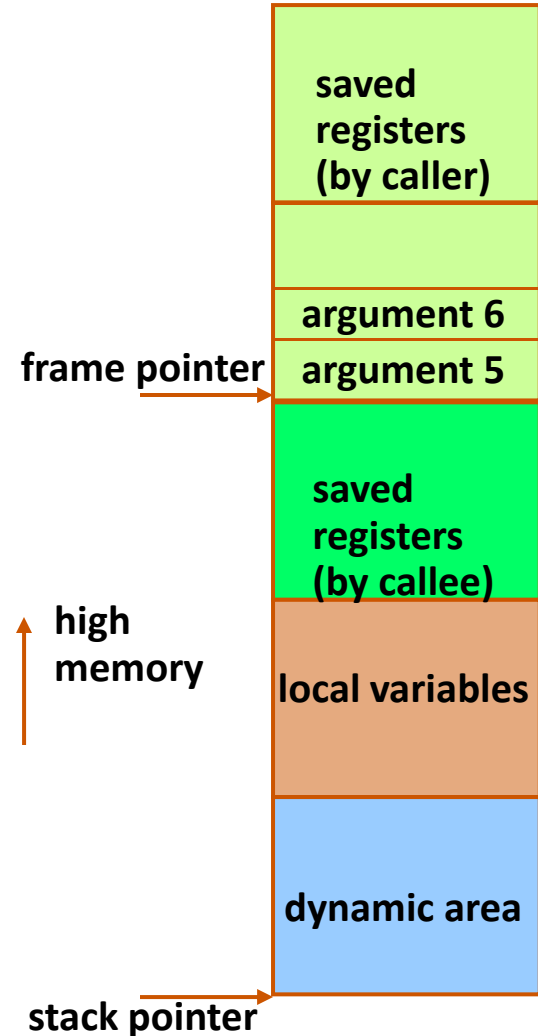
- **Why combine C and assembly language**
 - performance
 - C doesn't handle most architecture features, such as registers, program status, etc.
- **Develop C and assembly programs and then link them together**
 - at source level – in-line assembly code in C program
 - at object level – procedure call



Calling Convention

■ GCC calling convention

- 1st 4 arguments saved in registers
- registers saved by caller and callee (including frame pointer and returning PC)
- frame pointer points just below the last argument passed on the stack (the bottom of the frame)
- stack pointer points to the first word after the frame



APCS: ARM Procedure Call Standard

■ Constraints on the use of registers

■ Stack conventions

- **sp, fp, sl/v7, sb/v6, and v1-v5, must contain the same values when returning**

Register Number	APCS Name	APCS Role
0	a1	argument 1 / integer result / scratch register
1	a2	argument 2 / scratch register
2	a3	argument 3 / scratch register
3	a4	argument 4 / scratch register
4	v1	register variable
5	v2	register variable
6	v3	register variable
7	v4	register variable
8	v5	register variable
9	sb/v6	static base / register variable
10	sl/v7	stack limit / stack chunk handle / reg. variable
11	fp	frame pointer
12	ip	scratch register / new-sb in inter-link-unit calls
13	sp	lower end of current stack frame
14	lr	link address / scratch register
15	pc	program counter
f0	0	FP argument 1 / FP result / FP scratch register
f1	1	FP argument 2 / FP scratch register
f2	2	FP argument 3 / FP scratch register
f3	3	FP argument 4 / FP scratch register

Stack Backtrace Data Structure

save code pointer [fp, #0] ←fp points
 return link value [fp, #-4]
 return sp value [fp, #-8]
 return fp value [fp, #-12]
 {saved v7 value}
 {saved v6 value}
 {saved v5 value}
 {saved v4 value}
 {saved v3 value}
 {saved v2 value}
 {saved v1 value}
 {saved a4 value}
 {saved a3 value}
 {saved a2 value}
 {saved a1 value}

- Save code pointer (the value of pc) allows the function corresponding to a stack backtrace structure to be located

- Entry code

```
MOV      ip, sp      ; save current sp,
                    ; ready to save as old sp
STMFD   sp!, {a1-a4, v1-v5, sb, fp, ip, lr, pc}
                    ;as needed
```

```
SUB fp, ip, #4
```

- On function exit

```
return link value  pc
return sp value    sp
return fp value    fp
```

- If no use of stack, can be simple as

```
MOV      pc, lr      ; or
BX       lr
```

Calling Assembly Routine from C

- In C program**

```
char *srcstr = "First string - source ";
char *dststr = "Second string - destination ";
strcpy(dststr,srcstr);
```

- Assembly routine**

```
AREA SCopy, CODE, READONLY
EXPORT strcpy
```

strcpy

```
LDRB    r2, [r1],#1
STRB    r2, [r0],#1
CMP     r2, #0
BNE     strcpy
MOV     pc,lr
END
```

```
; r0 points to destination string.
; r1 points to source string.
; Load byte and update address.
; Store byte and update address
; Check for zero terminator.
; Keep going if not.
; Return.
```

Inline Assembly Code in C Program

- **A feature provided by C compiler**
 - compiler will do the insertion and knows the variables and the registers
- **Example: armcc**

```
void my_strcpy(char *src, char *dst)
{
    int ch;
    __asm
    {
        loop:
            LDRB ch, [src], #1
            STRB ch, [dst], #1
            CMP     ch, #0
            BNE     loop
    }
}
```

```
int main(void)
{
    char a[] = "Hello world!";
    char b[20];
    __asm
    {
        MOV     R0, a
        MOV     R1, b
        BL my_strcpy, {R0, R1}, {}, {}
    }
    printf("Original string: %s\n", a);
    printf("Copied string: %s\n", b);
    return 0;
}
```

Inline Assembly Language in armcc

- `__asm`

```
{
    instruction [; instruction]
    ...
    [ instruction]
}
```

- Use constant, register and variable operands, and C labels

- Use of physical registers ??

```
int bad_f(int x)           // x in r0
{
    __asm
    {
        ADD r0, r0, #1    // wrongly asserts
                          // that x is still in r0
    }
    return x;             // x in r0
}
```

- This will work

```
ADD x, x, #1
```

```
__inline void enable_IRQ(void)
```

```
{ int tmp;
  __asm
  {
      MRS    tmp, CPSR
      BIC    tmp, tmp, #0x80
      MSR    CPSR_c, tmp
  }
}
```

```
__inline void disable_IRQ(void)
```

```
{ int tmp;
  __asm
  {
      MRS    tmp, CPSR
      ORR    tmp, tmp, #0x80
      MSR    CPSR_c, tmp
  }
}
```

```
int main(void)
```

```
{
    disable_IRQ();    enable_IRQ();
}
```

Inline Assembly Language in gcc

- asm("code" : outputs : inputs : clobbers);

- Example 1**

```
asm("foo %1,%2,%0" : "=r" (output) : "r" (input1), "r" (input2));
```

The generated code could be

```
#APP
```

```
foo    r17,r5,r9    // %0,%1, and %2 are replaced with registers
                        // holding the first three argument.
```

```
#NO_APP
```

- Example 2**

```
asm("foo %1,%2,%0" : "=r" (ptr->vtable[3](a,b,c)->foo.bar[baz]) : : "r"
    (gcc(is) + really(damn->cool)), "r" (42));
```

GCC will treat this just like:

```
register int t0, t1, t2;
t1 = gcc(is) + really(damn->cool);
t2 = 42;
asm("foo %1,%2,%0" : "=r" (t0) : "r" (t1), "r" (t2));
ptr->vtable[3](a,b,c)->foo.bar[baz] = t0;
```

Example: C assignments

- **C:**
 $x = (a + b) - c;$
- **Assembler:**

```

ADR r4,a           ; get address for a
LDR r0,[r4]        ; get value of a
ADR r4,b           ; get address for b, reusing r4
LDR r1,[r4]        ; get value of b
ADD r3,r0,r1       ; compute a+b
ADR r4,c           ; get address for c
LDR r2,[r4]        ; get value of c
SUB r3,r3,r2       ; complete computation of x
ADR r4,x           ; get address for x
STR r3,[r4]        ; store value of x
        
```

Example: C assignment

- **C:**

```
y = a*(b+c);
```

- **Assembler:**

```
ADR r4,b ; get address for b
LDR r0,[r4] ; get value of b
ADR r4,c ; get address for c
LDR r1,[r4] ; get value of c
ADD r2,r0,r1 ; compute partial result
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MUL r2,r2,r0 ; compute final value for y
ADR r4,y ; get address for y
STR r2,[r4] ; store y
```

Example: C assignment

- **C:**

```
z = (a << 2) | (b & 15);
```

- **Assembler:**

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift
ADR r4,b ; get address for b
LDR r1,[r4] ; get value of b
AND r1,r1,#15 ; perform AND
ORR r1,r0,r1 ; perform OR
ADR r4,z ; get address for z
STR r1,[r4] ; store value for z
```

Example: if statement

- **C:**
`if (a > b) { x = 5; y = c + d; } else x = c - d;`

- **Assembler:**

`; compute and test condition`

`ADR r4,a ; get address for a`

`LDR r0,[r4] ; get value of a`

`ADR r4,b ; get address for b`

`LDR r1,[r4] ; get value for b`

`CMP r0,r1 ; compare a < b`

`BLE fblock ; if a >= b, branch to false block`

if statement, cont'd.

; true block

```
MOV r0,#5 ; generate value for x
ADR r4,x ; get address for x
STR r0,[r4] ; store x
ADR r4,c ; get address for c
LDR r0,[r4] ; get value of c
ADR r4,d ; get address for d
LDR r1,[r4] ; get value of d
ADD r0,r0,r1 ; compute y
ADR r4,y ; get address for y
STR r0,[r4] ; store y
B after ; branch around false block
```

if statement, cont'd.

; false block

fblock ADR r4,c ; get address for c

LDR r0,[r4] ; get value of c

ADR r4,d ; get address for d

LDR r1,[r4] ; get value for d

SUB r0,r0,r1 ; compute a-b

ADR r4,x ; get address for x

STR r0,[r4] ; store value of x

after ...

Example: Conditional instruction implementation

; true block

```

MOVLT r0,#5 ; generate value for x
ADRLT r4,x ; get address for x
STRLT r0,[r4] ; store x
ADRLT r4,c ; get address for c
LDRLT r0,[r4] ; get value of c
ADRLT r4,d ; get address for d
LDRLT r1,[r4] ; get value of d
ADDLT r0,r0,r1 ; compute y
ADRLT r4,y ; get address for y
STRLT r0,[r4] ; store y
    
```

Conditional instruction implementation, cont'd.

; false block

ADRGE r4,c ; get address for c

LDRGE r0,[r4] ; get value of c

ADRGE r4,d ; get address for d

LDRGE r1,[r4] ; get value for d

SUBGE r0,r0,r1 ; compute a-b

ADRGE r4,x ; get address for x

STRGE r0,[r4] ; store value of x

Example: switch statement

- **C:**

```
switch (test) { case 0: ... break; case 1: ... }
```
- **Assembler:**

```
ADR r2,test ; get address for test
LDR r0,[r2] ; load value for test
ADR r1,switchtab ; load address for switch table
LDR r1,[r1,r0,LSL #2] ; index switch table
switchtab DCD case0
          DCD case1
          ...
```

Example: FIR filter

- **C:**

```
for (i=0, f=0; i<N; i++)
    f = f + c[i]*x[i];
```

- **Assembler**

; loop initiation code

```
MOV r0,#0 ; use r0 for I
```

```
MOV r8,#0 ; use separate index for arrays
```

```
ADR r2,N ; get address for N
```

```
LDR r1,[r2] ; get value of N
```

```
MOV r2,#0 ; use r2 for f
```

FIR filter, cont'.d

```

ADR r3,c ; load r3 with base of c
ADR r5,x ; load r5 with base of x
; loop body
loop LDR r4,[r3,r8] ; get c[i]
    LDR r6,[r5,r8] ; get x[i]
    MUL r4,r4,r6 ; compute c[i]*x[i]
    ADD r2,r2,r4 ; add into running sum
    ADD r8,r8,#4 ; add one word offset to array index
    ADD r0,r0,#1 ; add 1 to i
    CMP r0,r1 ; exit?
    BLT loop ; if i < N, continue
    
```

Backup

Assembler: Pseudo-ops

AREA -> chunks of data (\$data) or code (\$code)

ADR -> load address into a register

ADR R0, BUFFER

ALIGN -> adjust location counter to word boundary usually after a storage directive

END -> no more to assemble

Assembler: Pseudo-ops

DCD -> defined word value storage area
BOW DCD 1024, 2055, 9051

DCB -> defined byte value storage area
BOB DCB 10, 12, 15

% -> zeroed out byte storage area
BLBYTE % 30

Assembler: Pseudo-ops

IMPORT -> name of routine to import for use in this routine

IMPORT _printf ; C print routine

EXPORT -> name of routine to export for use in other routines

EXPORT add2 ; add2 routine

EQU -> symbol replacement

loopcnt EQU 5

Assembly Line Format

label <whitespace> *instruction* <whitespace> ; *comment*

label: created by programmer, alphanumeric

whitespace: space(s) or tab character(s)

instruction: op-code mnemonic or pseudo-op with required fields

comment: preceded by ; ignored by assembler but useful
 to the programmer for documentation

NOTE: All fields are optional.

ARM Instruction Set Summary (1/4)

Mnemonic	Instruction	Action
ADC	Add with carry	$Rd := Rn + Op2 + Carry$
ADD	Add	$Rd := Rn + Op2$
AND	AND	$Rd := Rn \text{ AND } Op2$
B	Branch	$R15 := \text{address}$
BIC	Bit Clear	$Rd := Rn \text{ AND NOT } Op2$
BL	Branch with Link	$R14 := R15$ $R15 := \text{address}$
BX	Branch and Exchange	$R15 := Rn$ $T \text{ bit} := Rn[0]$
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	$CPSR \text{ flags} := Rn + Op2$
CMP	Compare	$CPSR \text{ flags} := Rn - Op2$

ARM Instruction Set Summary (2/4)

Mnemonic	Instruction	Action
EOR	Exclusive OR	$Rd := Rn \wedge Op2$
LDC	Load Coprocessor from memory	(Coprocessor load)
LDM	Load multiple registers	Stack Manipulation (Pop)
LDR	Load register from memory	$Rd := (address)$
MCR	Move CPU register to coprocessor register	$CRn := rRn \{<op> cRm\}$
MLA	Multiply Accumulate	$Rd := (Rm * Rs) + Rn$
MOV	Move register or constant	$Rd := Op2$
MRC	Move from coprocessor register to CPU register	$rRn := cRn \{<op> cRm\}$
MRS	Move PSR status/flags to register	$Rn := PSR$
MSR	Move register to PSR status/flags	$PSR := Rm$

ARM Instruction Set Summary (3/4)

Mnemonic	Instruction	Action
MUL	Multiply	$Rd := Rm * Rs$
MVN	Move negative register	$Rd := \sim Op2$
ORR	OR	$Rd := Rn \text{ OR } Op2$
RSB	Reverse Subtract	$Rd := Op2 - Rn$
RSC	Reverse Subtract with Carry	$Rd := Op2 - Rn - 1 + \text{Carry}$
SBC	Subtract with Carry	$Rd := Rn - Op2 - 1 + \text{Carry}$
STC	Store coprocessor register to memory	$\text{address} := cRn$
STM	Store Multiple	Stack manipulation (Push)

ARM Instruction Set Summary (4/4)

Mnemonic	Instruction	Action
STR	Store register to memory	<address>:=Rd
SUB	Subtract	Rd:=Rn-Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd:=[Rn] [Rn]:=Rm
TEQ	Test bitwise equality	CPSR flags:=Rn EOR Op2
TST	Test bits	CPSR flags:=Rn AND Op2