

ARM Assembly Programming

Computer Organization and Assembly Languages

Yung-Yu Chuang

with slides by Peng-Sheng Chen

GNU compiler and binutils

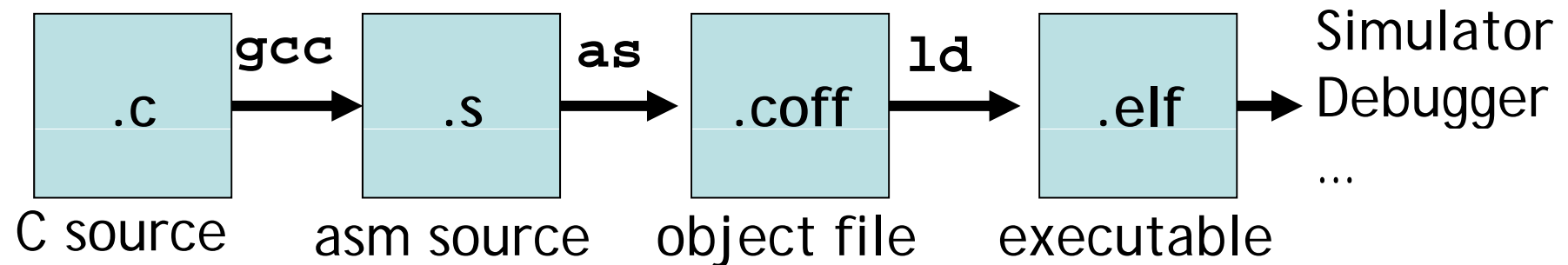


- GNU compiler and binutils
 - gcc: GNU C compiler
 - as: GNU assembler
 - ld: GNU linker
 - gdb: GNU project debugger
 - insight: a (Tcl/Tk) graphic interface to gdb

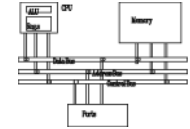
Pipeline



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



GAS program format

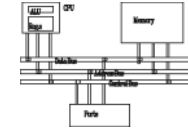


```
.file "test.s"
.text
.global main
.type main, %function

main:

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

GAS program format

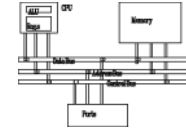


```
.file "test.s"
.text
export variable → .global main
.type main, %function
main:
MOV R0, #100
ADD R0, R0, R0
SWI #11
signals the end of the program → .end
```

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

call interrupt to end the program

ARM assembly program



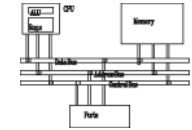
label	operation	operand	comments
<code>main:</code>	<code>LDR</code>	<code>R1, value</code>	<code>@ load value</code>
	<code>STR</code>	<code>R1, result</code>	
	<code>SWI</code>	<code>#11</code>	
<code>value:</code>	<code>.word</code>	<code>0x0000C123</code>	
<code>result:</code>	<code>.word</code>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

Control structures



- Program is to implement algorithms to solve problems. Program decomposition and flow of control are important concepts to express algorithms.
- Flow of control:
 - Sequence.
 - Decision: if-then-else, switch
 - Iteration: repeat-until, do-while, for
- Decomposition: split a problem into several smaller and manageable ones and solve them independently.
(subroutines/functions/procedures)

Decision



- If-then-else
- switch

If statements

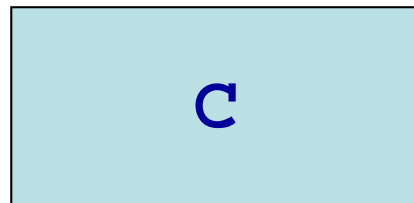


if **C** then **T** else **E**

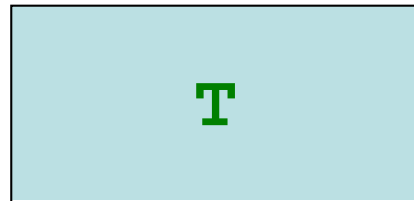
// find maximum

if (R0>R1) then R2:=R0

else R2:=R1

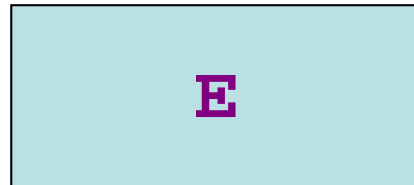


BNE else



B endif

else:



endif:

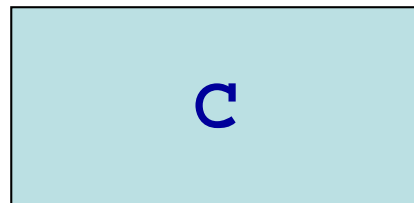
If statements



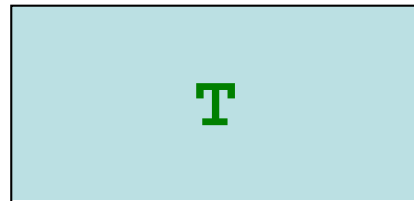
if **C** then **T** else **E**

// find maximum

if (R0>R1) then R2:=R0
else R2:=R1

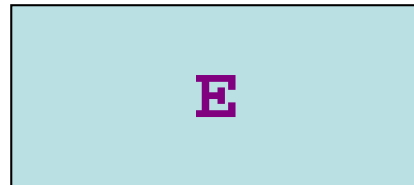


BNE else



B endif

else:



endif:

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
```

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

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

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

If statements



`R0=abs (R0)`

`CMP R0, #0`

`RSBMI R0, R0, #0`

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

The range is between 0 and N

Slow if N is large

Switch statements



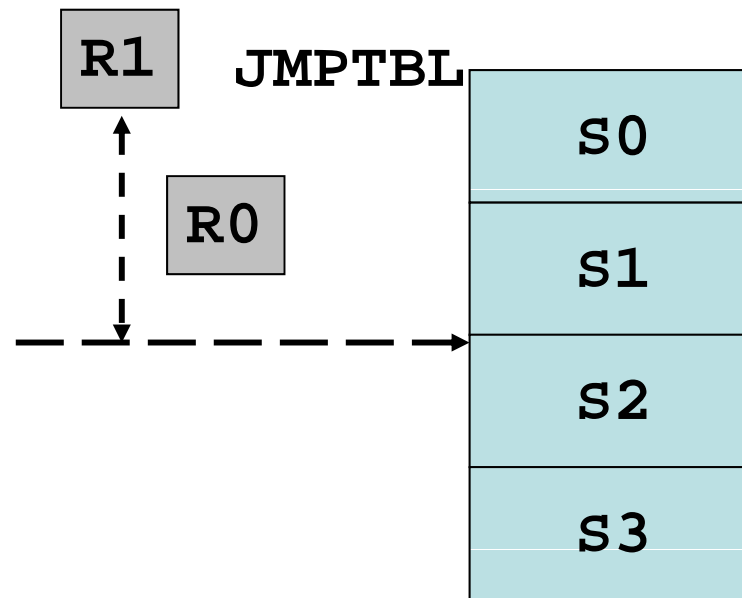
```
ADR    R1, JMPTBL  What if the range is between  
CMP    R0, #3      M and N?  
LDRLS  PC, [R1, R0, LSL #2]
```

err: ... For larger N and sparse values,
B exit we could use a hash function.

```
S0: ...
```

```
JMPTBL:
```

```
.word S0  
.word S1  
.word S2  
.word S3
```



Iteration



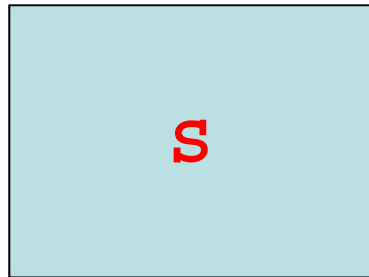
- repeat-until (do-while)
- while-do
- for

repeat loops



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

loop:



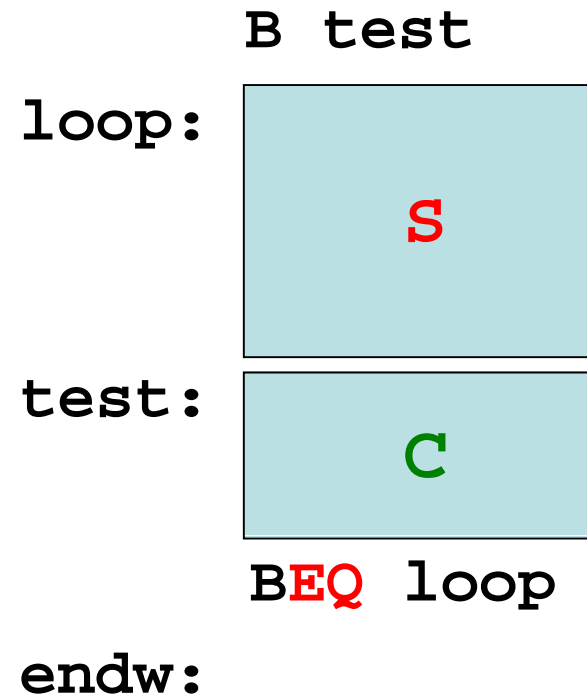
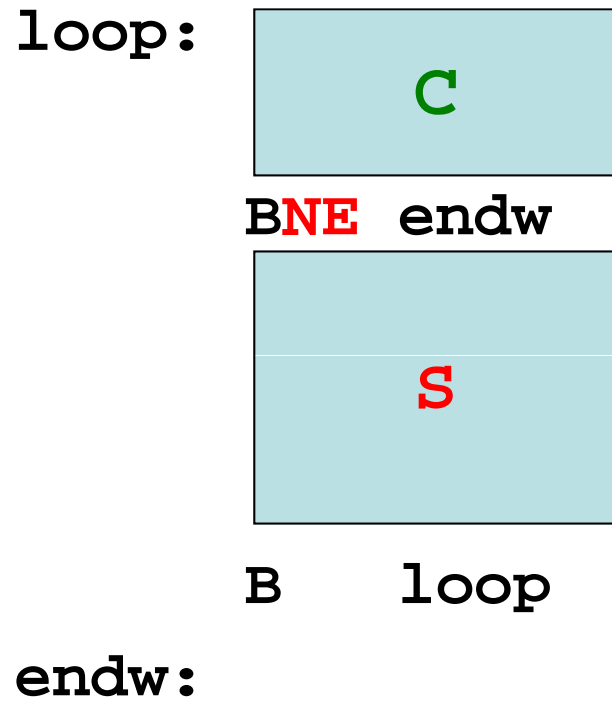
BEQ loop

endw:

while loops



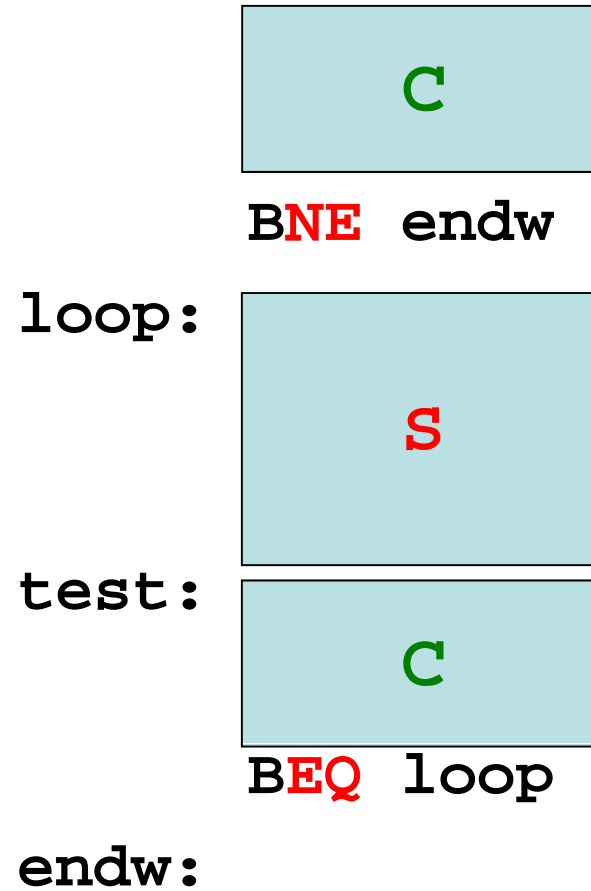
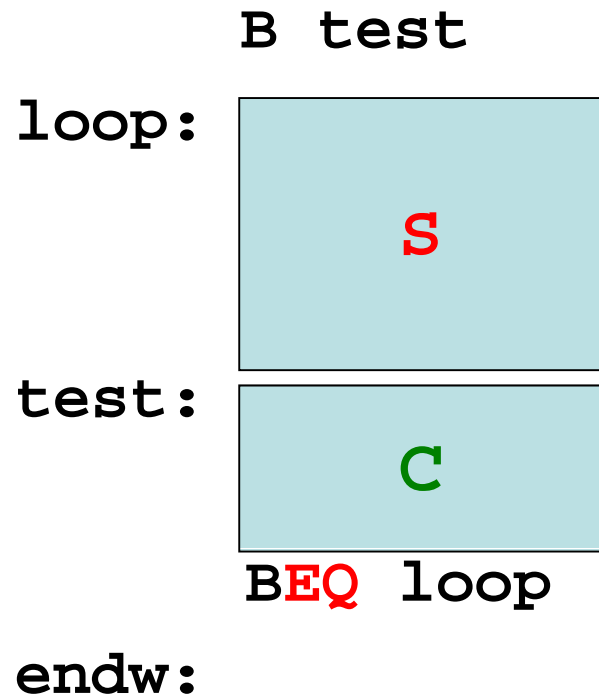
while (**C**) { **S** }



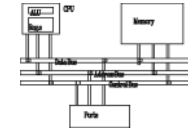
while loops



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



GCD



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

GCD



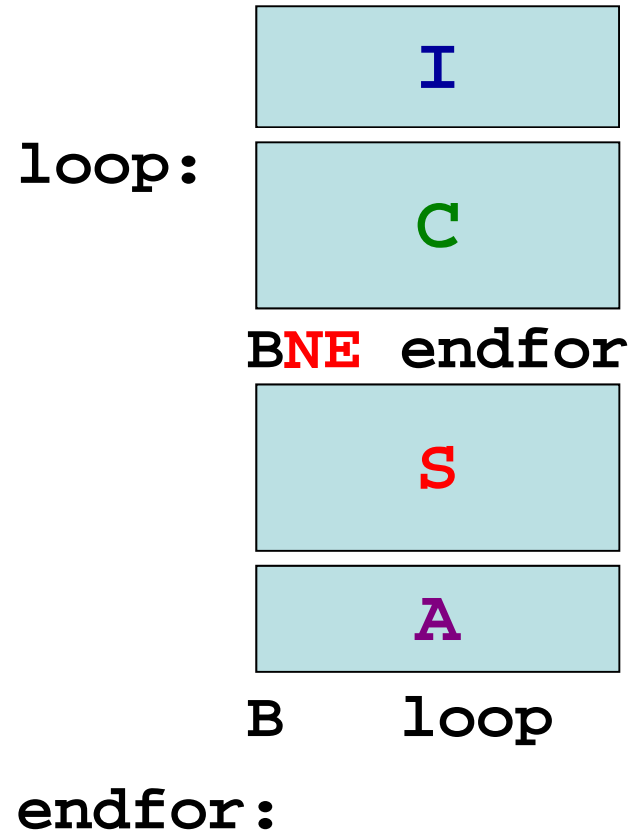
```
Loop:  CMP    R1, R2
        SUBGT  R1, R1, R2
        SUBLT  R2, R2, R1
        BNE   loop
```


for loops



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

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

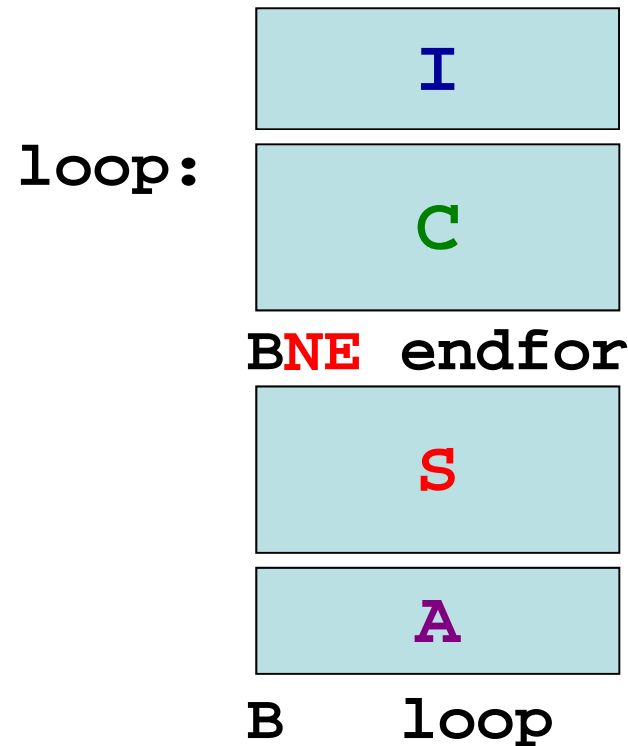


for loops



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

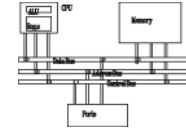
```
for (i=0; i<10; i++)  
{ a[i]:=0; }
```



endfor:

```
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

```
void func(int a, int b) callee
{
    ...
}
int main(void) caller
{
    func(100, 200);
    ...
}
```

↑ parameters

↓ arguments

Procedures



main:

```
...  
BL func → func:  
...  
...  
.end           .end
```

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

Procedures



main:

caller

callee

@ use R5

func:

BL func

...

@ use R5

@ use R5

...

...

...

...

.end

.end

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

Procedures (caller save)



main:

caller

callee

@ use R5

@ save R5

BL func

@ restore R5

@ use R5

.end

func:

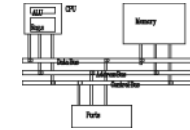
...

@ use R5

.end

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

Procedures (callee save)



main:

caller

@ use R5

BL func

@ use R5

.end

func: @ save R5

...

@ use R5

@restore R5

.end

callee

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

Procedures



main:

caller

callee

@ use R5

func:

BL func

...

@ use R5

@ use R5

...

...

...

...

.end

.end

- How to pass arguments? By registers? By stack? By memory? In what order?
- Who should save R5? Caller? Callee?
- We need a protocol for these.

ARM Procedure Call Standard (APCS)



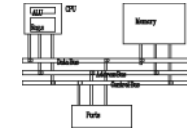
- ARM Ltd. defines a set of rules for procedure entry and exit so that
 - Object codes generated by different compilers can be linked together
 - Procedures can be called between high-level languages and assembly
- APCS defines
 - Use of registers
 - Use of stack
 - Format of stack-based data structure
 - 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

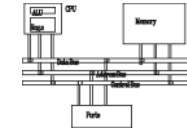
APCS register usage convention



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

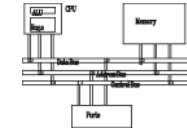
APCS register usage convention



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

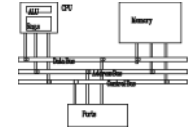
APCS register usage convention



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

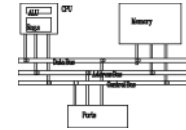
- Registers for special purposes
- Could be used as temporary variables if saved properly.

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)

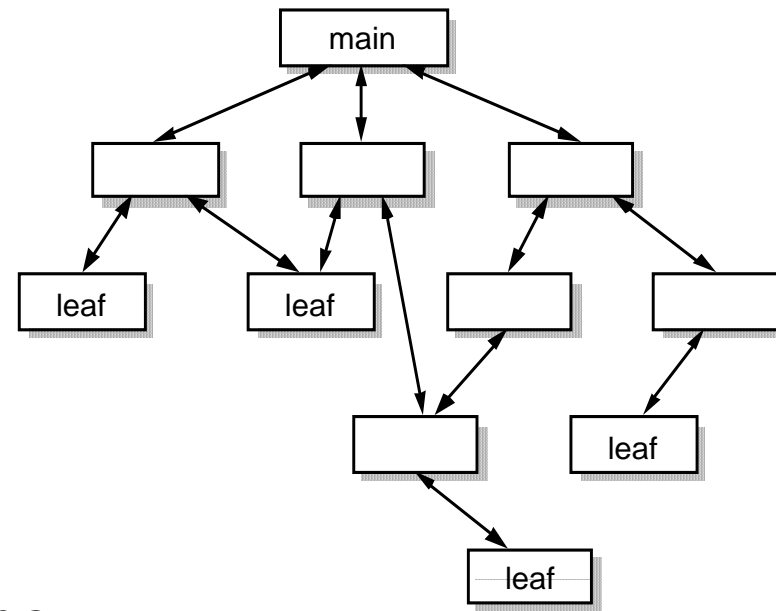
Function entry/exit



- 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
```

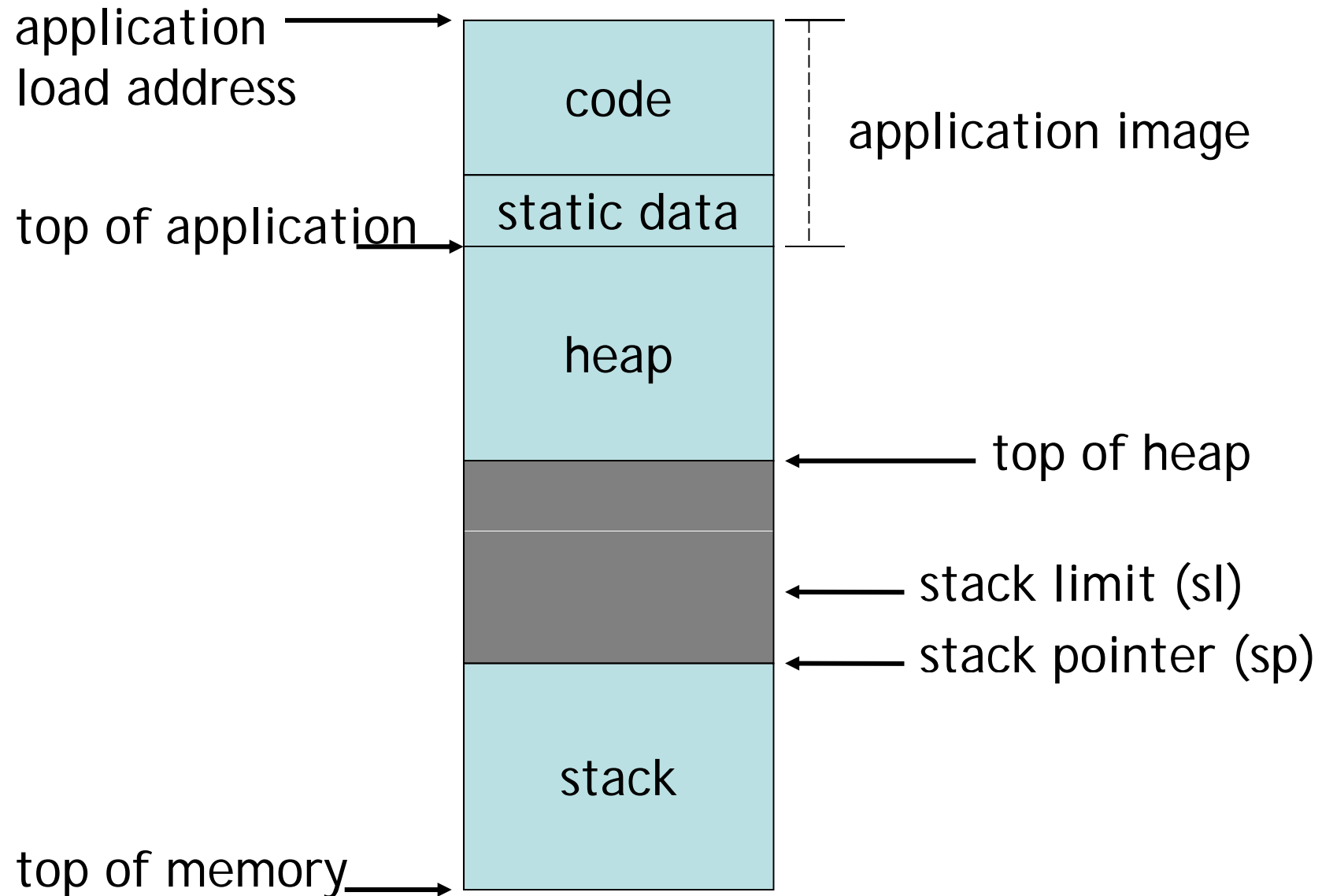
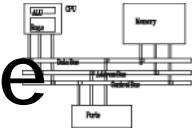
```
...
```

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

```
...
```

```
LDMFD sp!, {regs, pc} @ restore and  
@ return
```

Standard ARM C program address space

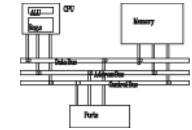


Accessing operands



- A procedure often accesses operands 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:

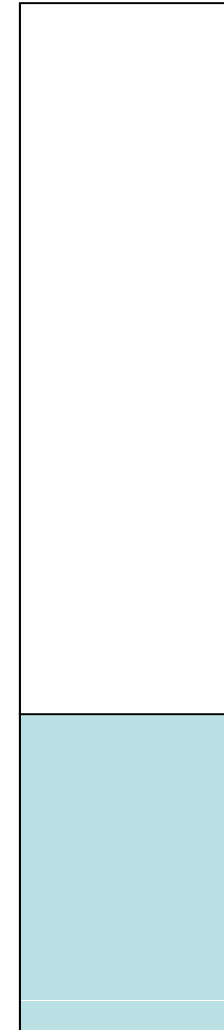
```
LDR    R0, #0
```

```
...
```

```
BL     func
```

```
...
```

low



high

stack

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}
```

