Instruction Set Architecture

Instructions

- Language for computer hardware
- Different computers may have different instruction sets
- However they are of similar nature
- Basic operations need to be supported
- Early computer had very simple instruction set

Assembly language for ARM

- Most popular in 32 bits
- Primarily used in embedded systems
- Belongs to RISC family
- Similar to MIPS

Arithmetic operation

- `ADD a, b, c`
  - Adds the two variables `b` and `c` and stores the result in `a`

Arithmetic operation

- `ADD a, b, c`
  - Adds the two variables `b` and `c` and stores the result in `a`
- Add four variables `b, c, d, e` and store the result in `a`

Arithmetic operation

- `ADD a, b, c`
  - Adds the two variables `b` and `c` and stores the result in `a`
- Add four variables `b, c, d, e` and store the result in `a`

```
ADD a, b, c
```

Arithmetic operation

- `ADD a, b, c`
  - Adds the two variables `b` and `c` and stores the result in `a`
- Add four variables `b, c, d, e` and store the result in `a`

```
ADD a, b, c
ADD a, a, d
```

Arithmetic operation

- `ADD a, b, c`
  - Adds the two variables `b` and `c` and stores the result in `a`
- Add four variables `b, c, d, e` and store the result in `a`

```
ADD a, b, c
ADD a, a, d
ADD a, a, e
```

Arithmetic operation (contd.)

- Two assignments: `a=b+c; d=a-e;`

Arithmetic operation (contd.)

- Two assignments: `a=b+c; d=a-e;`

```
ADD a, b, c
```

Arithmetic operation (contd.)

- Two assignments: `a=b+c; d=a-e;`

```
ADD a, b, c
SUB d, a, e
```

Arithmetic operation (contd.)

- Two assignments: `a=b+c; d=a-e;`

```
ADD a, b, c
SUB d, a, e
```

- Complex assignments: `f=(g+h)-(i+j)`

Arithmetic operation (contd.)

- Two assignments: `a=b+c; d=a-e;`

```
ADD a, b, c
SUB d, a, e
```

- Complex assignments: `f=(g+h)-(i+j)`

```
ADD t0, g, h
```

Arithmetic operation (contd.)

- Two assignments: `a=b+c; d=a-e;`

```
ADD a, b, c
SUB d, a, e
```

- Complex assignments: `f=(g+h)-(i+j)`

```
ADD t0, g, h
ADD t1, i, j
```

Arithmetic operation (contd.)

- Two assignments: `a=b+c; d=a-e;`

```
ADD a, b, c
SUB d, a, e
```

- Complex assignments: `f=(g+h)-(i+j)`

```
ADD t0, g, h
ADD t1, i, j
SUB f, t0, t1
```

Design principle - 1

- Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Operand for arithmetic operation

- Number of operand is restricted
- Uses special location ie. *register*
- For ARM size of the register is 32 bits
- Three operands of arithmetic instruction must be chosen from the registers

Design principle - 2

- Smaller is faster
  - Large number of registers increases the clock cycle time
  - Hardware cost
  - Trade off between cost and performance
  - To conserve energy

Arithmetic operation using registers

- Complex assignments: `f=(g+h)-(i+j)`
  - Let the variables `f, g, h, i, j` are assigned to register `r0, r1, r2, r3, r4`

Arithmetic operation using registers

- Complex assignments: `f=(g+h)-(i+j)`
  - Let the variables `f`, `g`, `h`, `i`, `j` are assigned to register `r0`, `r1`, `r2`, `r3`, `r4`
  - We need two temporary registers

Arithmetic operation using registers

- Complex assignments: `f=(g+h)-(i+j)`
  - Let the variables `f`, `g`, `h`, `i`, `j` are assigned to register `r0`, `r1`, `r2`, `r3`, `r4`
  - We need two temporary registers

```
ADD r5, r1, r2
```

Arithmetic operation using registers

- Complex assignments: `f=(g+h)-(i+j)`
  - Let the variables `f, g, h, i, j` are assigned to register `r0, r1, r2, r3, r4`
  - We need two temporary registers

```
ADD r5, r1, r2
ADD r6, r3, r4
```

Arithmetic operation using registers

- Complex assignments: `f=(g+h)-(i+j)`
  - Let the variables `f, g, h, i, j` are assigned to register `r0, r1, r2, r3, r4`
  - We need two temporary registers

```
ADD r5, r1, r2
ADD r6, r3, r4
SUB r0, r5, r6
```

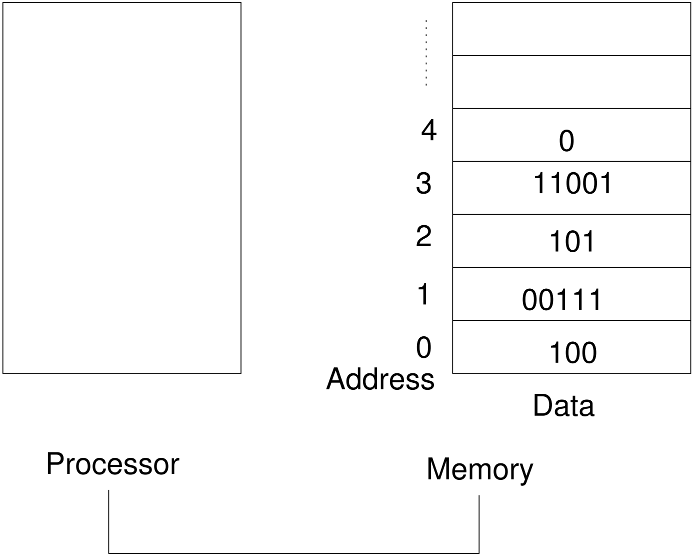
Operands from memory

- Complex data structures like `arrays` and `structures`
- All data may not be available in the registers
- The data are usually stored in memory

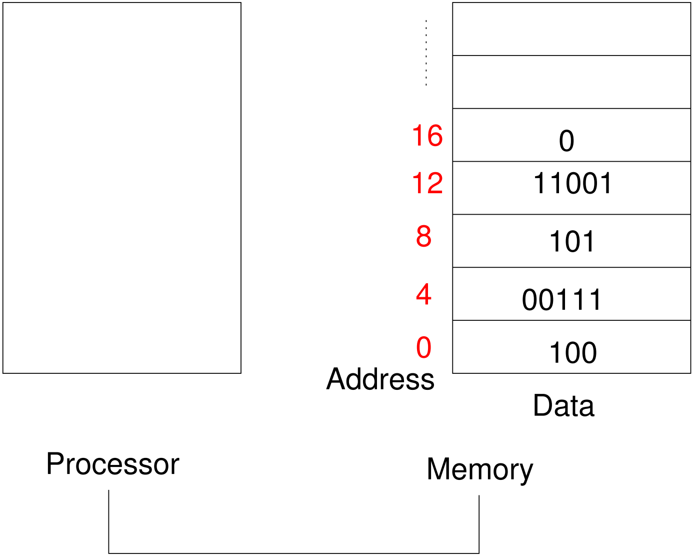
Operands from memory

- Complex data structures like `arrays` and `structures`
- All data may not be available in the registers
- The data are usually stored in memory
- How can a computer represent and access such data?

Memory addresses



Memory addresses for ARM



Arithmetic operation using operand from memory

- Assignments: `g=h+A[8]`
  - Let the variables `g`, `h` are assigned to register `r1`, `r2`
  - Let `r3` contains *base address* of array `A`
  - Need to use a temporary register `r5` (say) to store the data from memory

Arithmetic operation using operand from memory

- Assignments: `g=h+A[8]`
  - Let the variables `g, h` are assigned to register `r1, r2`
  - Let `r3` contains *base address* of array `A`
  - Need to use a temporary register `r5` (say) to store the data from memory

```
LDR r5, [r3,32]
```

Arithmetic operation using operand from memory

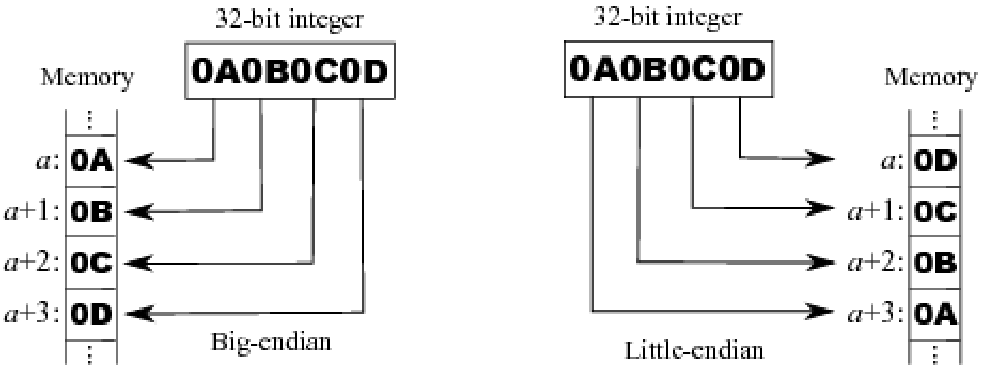
- Assignments: `g=h+A[8]`
  - Let the variables `g`, `h` are assigned to register `r1`, `r2`
  - Let `r3` contains *base address* of array `A`
  - Need to use a temporary register `r5` (say) to store the data from memory

```
LDR r5, [r3,32]
ADD r1, r2, r5
```

Memory

- Arrays and structures are allocated in memory
- Compiler places proper start address into data transfer function
- In ARM, memory is 32 bit wide (word)
- Each byte is addressable
- A *little-endian* machine stores the least significant byte first. ARM belongs to this group.

Little-endian vs Big-endian



Register vs. Memory

- Register are faster to access than memory
- Operating on memory data requires extra load/store call. More instructions get executed.
- Compiler tries to put most frequently data in register and rest are in memory
- To achieve highest performance and conserve energy, compiler must use the register efficiently

Constant or Immediate operand

- Add 4 to register 3
  - Need to load 4 in a register from memory

    ```
    LDR r5, [r1, #AddrConst4]
    ADD r3, r3, r5
    ```
- Extra load operation required

Constant or Immediate operand

- Add 4 to register 3
  - Need to load 4 in a register from memory

    ```
    LDR r5, [r1, #AddrConst4]
    ADD r3, r3, r5
    ```
- Extra load operation required
- To avoid such load, one of the operands of the instruction be a constant
  - `ADD r3, r3, #4`

Design principle - 3

- Make the common case fast
  - Constant operands occurs frequently
  - By including constants in arithmetic instruction extra call for load can be avoided
  - This will improve performance both in terms of time and power

Binary numbers

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

15 .... 3 2 1 0

MSB LSB

- $x15 \times 2^{15} + x14 \times 2^{14} + \ldots + x1 \times 2^{1} + x0 \times 2^{0}$
- How to handle signed number?
  - One bit may be reserved for sign, rest of the bits will denote the magnitude

Binary numbers

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

15 .... 3 2 1 0

MSB LSB

- $x15 \times 2^{15} + x14 \times 2^{14} + \ldots + x1 \times 2^{1} + x0 \times 2^{0}$
- How to handle signed number?
  - One bit may be reserved for sign, rest of the bits will denote the magnitude
    - Sign location - left or right?

Binary numbers

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
15	………											3	2	1	0
MSB															LSB

- $x15 \times 2^{15} + x14 \times 2^{14} + \ldots + x1 \times 2^{1} + x0 \times 2^{0}$
- How to handle signed number?
  - One bit may be reserved for sign, rest of the bits will denote the magnitude
    - Sign location - left or right?
    - How to handle ± 0?

Binary numbers

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

15 .... 3 2 1 0

MSB LSB

- $x15 \times 2^{15} + x14 \times 2^{14} + \ldots + x1 \times 2^{1} + x0 \times 2^{0}$
- How to handle signed number?
  - One bit may be reserved for sign, rest of the bits will denote the magnitude
    - Sign location - left or right?
    - How to handle ± 0?
    - How to set sign bit for add operation?

Binary numbers

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
15	…											3	2	1	0
MSB															LSB

- $x15 \times 2^{15} + x14 \times 2^{14} + \ldots + x1 \times 2^{1} + x0 \times 2^{0}$
- How to handle signed number?
  - One bit may be reserved for sign, rest of the bits will denote the magnitude
    - Sign location - left or right?
    - How to handle ± 0?
    - How to set sign bit for add operation?
    - Subtraction of a large number from a small one?

2s complement signed number

- Leading 0 means positive and leading 1 means negative number
- For 16 bit representation, 0 to $2^{15} - 1$ will be represented as before
- Bit pattern $1000\ldots000$ will be treated as -2^{15}
- Bit pattern $1111\ldots111$ will be treated as -1
- $x15 \times -2^{15} + x14 \times 2^{14} + \ldots + x1 \times 2^{1} + x0 \times 2^{0}$

Operation on 2s complement numbers

- Let X be binary number represented in 2s complement form. Find $X + \overline{X}$
- Negation of a number?
- Sign extension

Representation of Instruction

cond	F	I	Opcode	S	Rn	Rd	Operand2
4bits	2bits	1bit	4bits	1bit	4bits	4bits	12bits

- Cond – Related to conditional branch
- F – Different instruction format
- I – Immediate. If I is 1, second source is 12-bit immediate
- Opcode – Basic operation
- S – Set condition code
- Rn – First register source operand
- Rd – Destination register
- Operand2 – Second source operand

Example

- `ADD r5, r1, r2`

Example

- `ADD r5, r1, r2`

cond	F	I	Opcode	S	Rn	Rd	Operand2
14	0	0	4	0	1	5	2

Example

- `ADD r5, r1, r2`

cond	F	I	Opcode	S	Rn	Rd	Operand2
14	0	0	4	0	1	5	2

- `ADD r3, r3, #4`

Example

- `ADD r5, r1, r2`

cond	F	I	Opcode	S	Rn	Rd	Operand2
14	0	0	4	0	1	5	2

- `ADD r3, r3, #4`

cond	F	I	Opcode	S	Rn	Rd	Operand2
14	0	1	4	0	3	3	4

Data transfer instruction format

- Instruction format

Data transfer instruction format

- Instruction format

Cond	F	Opcode	Rn	Rd	Offset2
4bits	2bits	6bits	4bits	4bits	12bits

Data transfer instruction format

- Instruction format

Cond	F	Opcode	Rn	Rd	Offset2
4bits	2bits	6bits	4bits	4bits	12bits

- Example: `LDR r5, [r3,#32]`

Data transfer instruction format

- Instruction format

Cond	F	Opcode	Rn	Rd	Offset2
4bits	2bits	6bits	4bits	4bits	12bits

- Example: `LDR r5, [r3,#32]`

Cond	F	Opcode	Rn	Rd	Offset2
14	1	24	3	5	32

Assembly to binary

- Assignment operation: `A[30]=h+A[30]`
- Base addr of A is in r3 and h is in r1

Assembly to binary

- Assignment operation: `A[30]=h+A[30]`
- Base addr of A is in r3 and h is in r1

```
LDR r5, [r3, #120]
```

Assembly to binary

- Assignment operation: `A[30]=h+A[30]`
- Base addr of A is in r3 and h is in r1

```
LDR r5, [r3, #120]
ADD r5, r1, r5
```

Assembly to binary

- Assignment operation: `A[30]=h+A[30]`
- Base addr of A is in r3 and h is in r1

```
LDR r5, [r3, #120]
ADD r5, r1, r5
STR r5, [r3, #120]
```

Assembly to binary

- Assignment operation: `A[30]=h+A[30]`
- Base addr of A is in r3 and h is in r1

```
LDR r5, [r3, #120]
ADD r5, r1, r5
STR r5, [r3, #120]
```

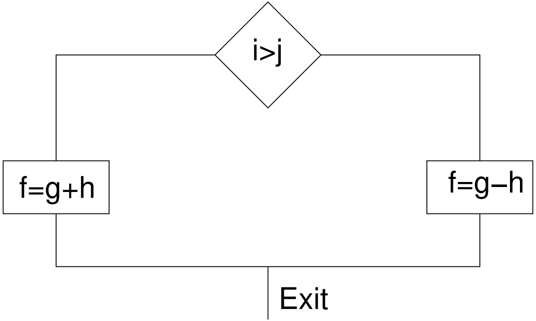
- Binary code

Cond	F	opcode			Rn	Rs	offset2
		I	opcode	S			operand2
14	1	24			3	5	120
14	0	0	4	0	2	5	5
14	1	25			3	5	120

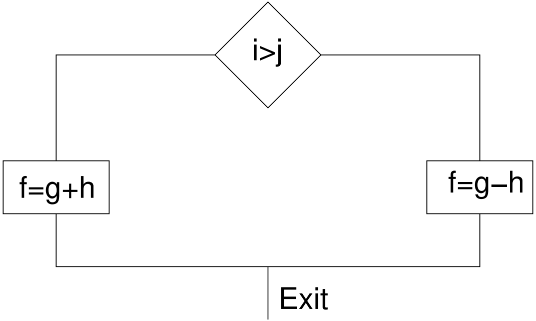
Logical operation

- Bit-by-bit AND – `AND r5, r1, r2`
- Bit-by-bit OR – `ORR r5, r1, r2`
- Bit-by-bit NOT – `MVN r5, r1`
- Shift left – `LSL`
  - `r5 = r1 + (r2<<2)`
  - `ADD r5, r1, r2, LSL #2`
- Shift right – `LSR`
  - `r6 = r5>>r3`
  - `MOV r6, r5, LSR r3`
- No separate instruction for logical shift. It is part of data processing instruction.

Instructions for making decision



Instructions for making decision




```
CMP register1, register2
BEQ L1
BNE L2
```

If-then-else

- `if(i==j) f=g+h; else f=g-h;`
- `f, g, h, i, j` are in `r0` to `r4`

If-then-else

- `if(i==j) f=g+h; else f=g-h;`
- `f, g, h, i, j` are in `r0 to r4`

```
CMP r3, r4
```

If-then-else

- `if(i==j) f=g+h; else f=g-h;`
- `f, g, h, i, j` are in `r0 to r4`

```
CMP r3, r4
BNE Else
```

If-then-else

- `if(i==j) f=g+h; else f=g-h;`
- `f, g, h, i, j` are in `r0 to r4`

```
CMP r3, r4
BNE Else
ADD r0, r1, r2
```

If-then-else

- `if(i==j) f=g+h; else f=g-h;`
- `f, g, h, i, j` are in `r0` to `r4`

```
CMP r3, r4
BNE Else
ADD r0, r1, r2
B Exit
```

If-then-else

- `if(i==j) f=g+h; else f=g-h;`
- `f, g, h, i, j` are in `r0 to r4`

```
CMP r3, r4
BNE Else
ADD r0, r1, r2
B Exit
Else:  SUB r0, r1, r2
```

If-then-else

- `if(i==j) f=g+h; else f=g-h;`
- `f, g, h, i, j` are in `r0 to r4`

```
CMP r3, r4
BNE Else
ADD r0, r1, r2
B Exit
Else:  SUB r0, r1, r2
Exit
```

Loops

- `while(save[i] == k) i += 1;`
- `i-r3, k-r5, save-r6`

Loops

- `while(save[i] == k) i += 1;`
- `i-r3, k-r5, save-r6`

```
Loop:  ADD r12, r6, r3, LSL #2
```

Loops

- `while(save[i] == k) i += 1;`
- `i-r3, k-r5, save-r6`

```
Loop:  ADD r12, r6, r3, LSL #2
LDR r0, [r12,#0]
```

Loops

- `while(save[i] == k) i += 1;`
- `i-r3, k-r5, save-r6`

```
Loop:  ADD r12, r6, r3, LSL #2
LDR r0, [r12,#0]
CMP r0, r5
```

Loops

- `while(save[i] == k) i += 1;`
- `i-r3, k-r5, save-r6`

```
Loop:  ADD r12, r6, r3, LSL #2
LDR r0, [r12,#0]
CMP r0, r5
BNE Exit
```

Loops

- `while(save[i] == k) i += 1;`
- `i-r3, k-r5, save-r6`

```
Loop:  ADD r12, r6, r3, LSL #2
LDR r0, [r12,#0]
CMP r0, r5
BNE Exit
ADD r3, r3, #1
```

Loops

- `while(save[i] == k) i += 1;`
- `i-r3, k-r5, save-r6`

```
Loop:  ADD r12, r6, r3, LSL #2
LDR r0, [r12,#0]
CMP r0, r5
BNE Exit
ADD r3, r3, #1
B Loop
```

Loops

- `while(save[i] == k) i += 1;`
- `i-r3, k-r5, save-r6`

```
Loop:  ADD r12, r6, r3, LSL #2
LDR r0, [r12,#0]
CMP r0, r5
BNE Exit
ADD r3, r3, #1
B Loop
Exit
```

Signed and unsigned comparison

- `r0=1111 1111 1111 1111`
- `r1=0000 0000 0000 0001`
- `CMP r0, r1`

Signed and unsigned comparison

- `r0=1111 1111 1111 1111`
- `r1=0000 0000 0000 0001`
- ```
  CMP r0, r1
      BLO L1 ; unsigned branch
  ```
  ```

Signed and unsigned comparison

- `r0=1111 1111 1111 1111`
- `r1=0000 0000 0000 0001`
- `CMP r0, r1`

```
BLO L1 ; unsigned branch
BLT L2 ; signed branch
```

Encoding of branch instruction

Cond	12	address
4bits	4bits	24 bits

Encoding of branch instruction

Cond	12	address
4bits	4bits	24 bits

- Cond can be `EQ`, `NE`, `HS`, ... , `LE`, `GE`

Encoding of branch instruction

Cond	12	address
4bits	4bits	24 bits

- Cond can be `EQ`, `NE`, `HS`, ... , `LE`, `GE`
- address is 24 bit wide. Memory of 16MB can be addressed.

Encoding of branch instruction

Cond	12	address
4bits	4bits	24 bits

- Cond can be `EQ`, `NE`, `HS`, ... , `LE`, `GE`
- address is 24 bit wide. Memory of 16MB can be addressed.
- Program counter = Register + Branch Address

Encoding of branch instruction

Cond	12	address
4bits	4bits	24 bits

- Cond can be `EQ`, `NE`, `HS`, ... , `LE`, `GE`
- address is 24 bit wide. Memory of 16MB can be addressed.
- Program counter = Register + Branch Address
- PC is chosen as register!

Encoding of branch instruction

Cond	12	address
4bits	4bits	24 bits

- Cond can be `EQ`, `NE`, `HS`, ... , `LE`, `GE`
- address is 24 bit wide. Memory of 16MB can be addressed.
- Program counter = Register + Branch Address
- PC is chosen as register!
- This branch addressing is called *PC-relative addressing*

Conditional execution: if-then-else

- Consider the same if-else statement

```
CMP r3, r4
BNE Else
ADD r0, r1, r2
B Exit
Else:SUB r0, r1, r2
Exit:
```

Conditional execution: if-then-else

- Consider the same if-else statement

```
CMP r3, r4
BNE Else
ADD r0, r1, r2
B Exit
Else:SUB r0, r1, r2
Exit:
```

```
CMP r3, r4
ADDEQ r0, r1, r2
SUBNE r0, r1, r2
```

Procedures

Procedures

- Put the parameters in a place where procedure can access

Procedures

- Put the parameters in a place where procedure can access
- Transfer control to procedure

Procedures

- Put the parameters in a place where procedure can access
- Transfer control to procedure
- Acquire storage resources needed for it

Procedures

- Put the parameters in a place where procedure can access
- Transfer control to procedure
- Acquire storage resources needed for it
- Perform desired task

Procedures

- Put the parameters in a place where procedure can access
- Transfer control to procedure
- Acquire storage resources needed for it
- Perform desired task
- Put return value in a place where the calling program can access it

Procedures

- Put the parameters in a place where procedure can access
- Transfer control to procedure
- Acquire storage resources needed for it
- Perform desired task
- Put return value in a place where the calling program can access it
- Return control to the point of origin

Handling of procedures in ARM

Handling of procedures in ARM

- r0–r3, r12 : Used to pass arguments and results (scratch registers)

Handling of procedures in ARM

- r0–r3, r12 : Used to pass arguments and results (scratch registers)
- lr (r14) : Link register contains the return address to return to point of origin

Handling of procedures in ARM

- r0–r3, r12 : Used to pass arguments and results (scratch registers)
- lr (r14) : Link register contains the return address to return to point of origin
- PC (r15) : Program counter

Handling of procedures in ARM

- r0–r3, r12 : Used to pass arguments and results (scratch registers)
- lr (r14) : Link register contains the return address to return to point of origin
- PC (r15) : Program counter
- `BL ProcedureAddress`

Handling of procedures in ARM

- r0–r3, r12 : Used to pass arguments and results (scratch registers)
- lr (r14) : Link register contains the return address to return to point of origin
- PC (r15) : Program counter
- `BL ProcedureAddress`
- SP (r13) : Spilling register organized as last-in-first-out queue

Handling of procedures in ARM

- r0–r3, r12 : Used to pass arguments and results (scratch registers)
- lr (r14) : Link register contains the return address to return to point of origin
- PC (r15) : Program counter
- `BL ProcedureAddress`
- SP (r13) : Spilling register organized as last-in-first-out queue
- `MOV pc, lr` – Unconditional jump

Example

```
int leaf_examp(int g, int h, int i, int j)
{
    int f;
    f = (g+h)-(i+j);
    return f;
}
```

Example

```
int leaf_examp(int g, int h, int i, int j)
{
     int f;
     f = (g+h)-(i+j);
     return f;
}
```

```
Leaf_examp:

ADD r5, r0, r1
ADD r6, r2, r3
SUB r4, r5, r6
```

Example

```
int leaf_examp(int g, int h, int i, int j)
{
    int f;
    f = (g+h)-(i+j);
    return f;
}
```

```
Leaf_examp:

ADD r5, r0, r1
ADD r6, r2, r3
SUB r4, r5, r6
MOV r0, r4
```

Example

```
int leaf_examp(int g, int h, int i, int j)
{
    int f;
    f = (g+h)-(i+j);
    return f;
}
```

```
Leaf_examp:
SUB sp, sp, #12
STR r6, [sp,#8]
STR r5, [sp,#4]
STR r4, [sp,#0]
ADD r5, r0, r1
ADD r6, r2, r3
SUB r4, r5, r6
MOV r0, r4
```

Example

```c
int leaf_examp(int g, int h, int i, int j)
{
    int f;
    f = (g+h)-(i+j);
    return f;
}
```

```
Leaf_examp:
SUB sp, sp, #12
STR r6, [sp,#8]
STR r5, [sp,#4]
STR r4, [sp,#0]
ADD r5, r0, r1
ADD r6, r2, r3
SUB r4, r5, r6
MOV r0, r4
LDR r4, [sp,#0]
LDR r5, [sp,#4]
LDR r6, [sp,#8]
ADD sp, sp, #12
```

Example

```c
int leaf_examp(int g, int h, int i, int j)
{
    int f;
    f = (g+h)-(i+j);
    return f;
}
```

```
Leaf_examp:
SUB sp, sp, #12
STR r6, [sp,#8]
STR r5, [sp,#4]
STR r4, [sp,#0]
ADD r5, r0, r1
ADD r6, r2, r3
SUB r4, r5, r6
MOV r0, r4
LDR r4, [sp,#0]
LDR r5, [sp,#4]
LDR r6, [sp,#8]
ADD sp, sp, #12
MOV pc, lr
```

Nested procedure

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

Nested procedure

```
fact:
SUB sp, sp, #8
STR lr, [sp,#4]
STR r0, [sp,#0]
```

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

Nested procedure

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

```
fact:
SUB sp, sp, #8
STR lr, [sp,#4]
STR r0, [sp,#0]
CMP r0,#1
BGE L1
```

Nested procedure

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

```
fact:
SUB sp, sp, #8
STR lr, [sp,#4]
STR r0, [sp,#0]
CMP r0,#1
BGE L1
MOV r0, #1
ADD sp, sp, #8
MOV pc, lr
```

Nested procedure

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

```
fact:
SUB sp, sp, #8
STR lr, [sp,#4]
STR r0, [sp,#0]
CMP r0,#1
BGE L1
MOV r0, #1
ADD sp, sp, #8
MOV pc, lr
L1:  SUB r0, r0, #1
BL fact
```

Nested procedure

```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

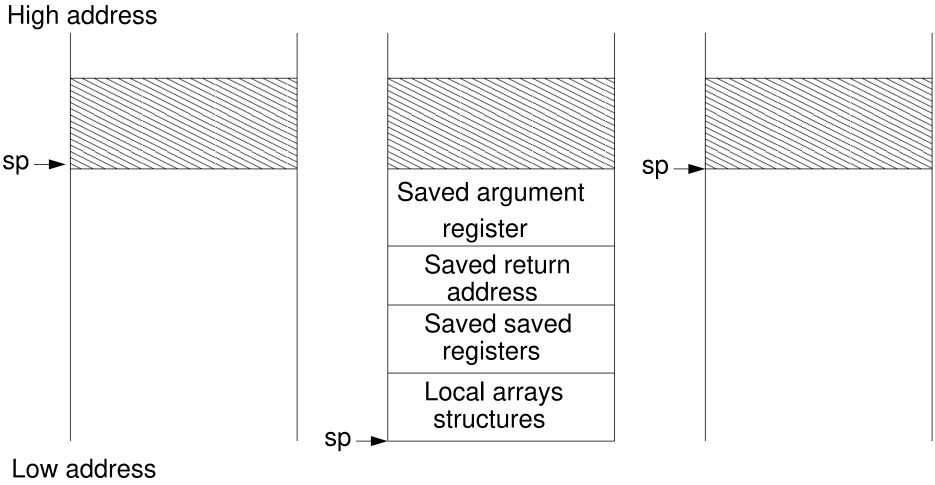
```
fact:
SUB sp, sp, #8
STR lr, [sp,#4]
STR r0, [sp,#0]
CMP r0,#1
BGE L1
MOV r0, #1
ADD sp, sp, #8
MOV pc, lr
L1:  SUB r0, r0, #1
BL fact
MOV r12, r0
LDR r0, [sp,#0]
LDR lr, [sp,#4]
ADD sp, sp, #8
```

Nested procedure

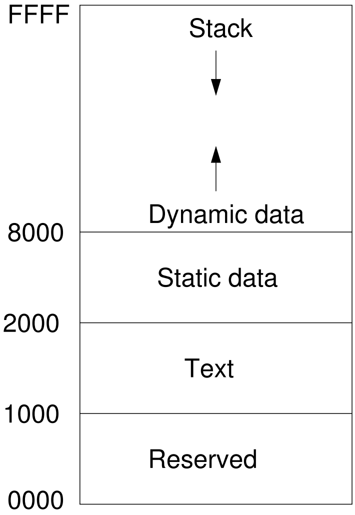
```
int fact(int n)
{
   if(n<1)
     return 1;
   else
     return (n*fact(n-1));
}
```

```
fact:
SUB sp, sp, #8
STR lr, [sp,#4]
STR r0, [sp,#0]
CMP r0,#1
BGE L1
MOV r0, #1
ADD sp, sp, #8
MOV pc, lr
L1:  SUB r0, r0, #1
BL fact
MOV r12, r0
LDR r0, [sp,#0]
LDR lr, [sp,#4]
ADD sp, sp, #8
MUL r0, r0, r12
MOV pc, lr
```

Stack allocation



Memory allocation for data & program



Load/Store a byte

- `LDRB r0, [r3,#0]` – Loads a byte from the memory and places it in the rightmost 8 bits of register
- `STRB r0, [r3,#0]` – Stores a byte from the register (rightmost 8 bits) into the memory location
- Sign extension?

strcpy

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- R0, y -- R1
```

strcpy

```
strcpy:
```

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- R0, y -- R1
```

strcpy

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- R0, y -- R1
```

```
strcpy:


MOV r4, #0
```

strcpy

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- R0, y -- R1
```

```
strcpy:


MOV r4, #0
L1:  ADD r2, r4, r1
LDRB r3, [r2,#0]
```

strcpy

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- R0, y -- R1
```

```
strcpy:


MOV r4, #0
L1:  ADD r2, r4, r1
LDRB r3, [r2,#0]
ADD r12, r4, r0
STRB r3, [r12,#0]
CMP r3, 0
```

strcpy

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- R0, y -- R1
```

```
strcpy:

MOV r4, #0
L1:  ADD r2, r4, r1
LDRB r3, [r2,#0]
ADD r12, r4, r0
STRB r3, [r12,#0]
CMP r3, 0
BEQ L2
ADD r4, r4, #1
B L1
```

strcpy

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}

x -- R0, y -- R1
```

```
strcpy:
SUB sp, sp, #4
STR r4, [sp,#0]
MOV r4, #0
L1:  ADD r2, r4, r1
LDRB r3, [r2,#0]
ADD r12, r4, r0
STRB r3, [r12,#0]
CMP r3, 0
BEQ L2
ADD r4, r4, #1
B L1
```

strcpy

```
void strcpy(char x[], char y[])
{
   int i;
   i = 0;
   while((x[i]=y[i])!= '\0')
     i++;
}
```

x -- R0, y -- R1

```
strcpy:
SUB sp, sp, #4
STR r4, [sp,#0]
MOV r4, #0
L1:  ADD r2, r4, r1
LDRB r3, [r2,#0]
ADD r12, r4, r0
STRB r3, [r12,#0]
CMP r3, 0
BEQ L2
ADD r4, r4, #1
B L1
L2:  LDR r4, [sp,#0]
ADD sp, sp, #4
MOV pc, lr
```

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`
- Scaled Register – `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`
- Scaled Register – `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`
- PC relative – `BEQ 1000 ; addr = PC + 1000`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`
- Scaled Register – `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`
- PC relative – `BEQ 1000 ; addr = PC + 1000`
- Immediate offset – `LDR r2, [r0,#8]`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`
- Scaled Register – `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`
- PC relative – `BEQ 1000 ; addr = PC + 1000`
- Immediate offset – `LDR r2, [r0,#8]`
  - `r2 = M[r0 + 8]`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`
- Scaled Register – `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`
- PC relative – `BEQ 1000 ; addr = PC + 1000`
- Immediate offset – `LDR r2, [r0,#8]`
  - `r2 = M[r0 + 8]`
- Register offset – `LDR r2, [r0,r1]`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`
- Scaled Register – `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`
- PC relative – `BEQ 1000 ; addr = PC + 1000`
- Immediate offset – `LDR r2, [r0,#8]`
  - `r2 = M[r0 + 8]`
- Register offset – `LDR r2, [r0,r1]`
  - `r2 = M[r0 + r1]`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`
- Scaled Register – `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`
- PC relative – `BEQ 1000 ; addr = PC + 1000`
- Immediate offset – `LDR r2, [r0,#8]`
  - `r2 = M[r0 + 8]`
- Register offset – `LDR r2, [r0,r1]`
  - `r2 = M[r0 + r1]`
- Scaled register offset – `LDR r2, [r0,r1,LSL #2]`

Addressing mode

- Immediate – `ADD r2, r0, #5 ; r2=r0+5`
- Register – `ADD r2, r0, r1 ; r2=r0+r1`
- Scaled Register – `ADD r2, r0, r1, LSL #2 ; r2=r0+(r1<<2)`
- PC relative – `BEQ 1000 ; addr = PC + 1000`
- Immediate offset – `LDR r2, [r0,#8]`
  - `r2 = M[r0 + 8]`
- Register offset – `LDR r2, [r0,r1]`
  - `r2 = M[r0 + r1]`
- Scaled register offset – `LDR r2, [r0,r1,LSL #2]`
  - `r2 = M[r0 + (r1<<2)]`

Addressing mode

- Immediate offset pre-index – `LDR r2, [r0,#4]!`

Addressing mode

- Immediate offset pre-index – `LDR r2, [r0,#4]!`
  - `r2 = M[ r0 + 4]`
  - `r0 = r0 + 4`

Addressing mode

- Immediate offset pre-index – `LDR r2, [r0,#4]!`
  - `r2 = M[ r0 + 4]`
  - `r0 = r0 + 4`
- Immediate offset post-index – `LDR r2, [r0], #4`

Addressing mode

- Immediate offset pre-index – `LDR r2, [r0,#4]!`
  - `r2 = M[ r0 + 4]`
  - `r0 = r0 + 4`
- Immediate offset post-index – `LDR r2, [r0], #4`
  - `r2 = M[r0]`
  - `r0 = r0 + 4`

Addressing mode

- Immediate offset pre-index – `LDR r2, [r0,#4]!`
  - `r2 = M[ r0 + 4]`
  - `r0 = r0 + 4`
- Immediate offset post-index – `LDR r2, [r0], #4`
  - `r2 = M[r0]`
  - `r0 = r0 + 4`
- Register offset pre-index – `LDR r2, [r0,r1]!`

Addressing mode

- Immediate offset pre-index – `LDR r2, [r0,#4]!`
  - `r2 = M[ r0 + 4]`
  - `r0 = r0 + 4`
- Immediate offset post-index – `LDR r2, [r0], #4`
  - `r2 = M[r0]`
  - `r0 = r0 + 4`
- Register offset pre-index – `LDR r2, [r0,r1]!`
  - `r2 = M[r0+r1]`
  - `r0 = r0 + r1`

Addressing mode

- Immediate offset pre-index – `LDR r2, [r0,#4]!`
  - `r2 = M[ r0 + 4]`
  - `r0 = r0 + 4`
- Immediate offset post-index – `LDR r2, [r0], #4`
  - `r2 = M[r0]`
  - `r0 = r0 + 4`
- Register offset pre-index – `LDR r2, [r0,r1]!`
  - `r2 = M[r0+r1]`
  - `r0 = r0 + r1`
- Register offset post-index – `LDR r2, [r0], r1`

Addressing mode

- Immediate offset pre-index – `LDR r2, [r0,#4]!`
  - `r2 = M[ r0 + 4]`
  - `r0 = r0 + 4`
- Immediate offset post-index – `LDR r2, [r0], #4`
  - `r2 = M[r0]`
  - `r0 = r0 + 4`
- Register offset pre-index – `LDR r2, [r0,r1]!`
  - `r2 = M[r0+r1]`
  - `r0 = r0 + r1`
- Register offset post-index – `LDR r2, [r0], r1`
  - `r2 = M[r0]`
  - `r0 = r0 + r1`

Addressing mode

- Scaled register offset pre-index – `LDR r2, [r0, r1, LSL #2]!`

Addressing mode

- Scaled register offset pre-index – `LDR r2, [r0, r1, LSL #2]!`
  - `r2 = M[ r0 + (r1<<2)]`
  - `r0 = r0 + (r1<<2)`

32 bit constant

- Most constant are small
- 12 bit operand2 field is divided into two fields:
  - 8 bit constant field on the right
  - 4 bit rotate right
  - Number can be represented in this format $X \times 2^{2i}$ where X lies between 0–255 and i is between 0–15

Arrays vs. Pointer

```
void clear1(int array[], int n)
{
    int i;
    for(i=0; i<n; i++)
      array[i]=0;
}
```

```
void clear2(int *ar, int n)
{
    int *p;
    for(p=&ar[0]; p<&ar[n]; p++)
      *p=0;
}
```

Arrays vs. Pointer

```
void clear1(int array[], int n)
{
   int i;
   for(i=0; i<n; i++)
     array[i]=0;
}

array -- R0, n -- R1
i -- R2, zero -- R3
```

Arrays vs. Pointer

```
void clear1(int array[], int n)
{
   int i;
   for(i=0; i<n; i++)
     array[i]=0;
}

array -- R0, n -- R1
i -- R2, zero -- R3
```

```
MOV i, 0
MOV zero, 0
loop1:
STR zero, [array, i, LSL #2]
ADD i, i, #1
CMP i, n
BLT loop1
```

Arrays vs. Pointer

```
void clear2(int *array, int n){
   int *p;
   for(p=&array[0];
     p<&array[n];
     p++)
     *p=0;
}

array -- R0, n -- R1
p -- R2, zero -- R3
arraySize -- R12
```

Arrays vs. Pointer

```
void clear2(int *array, int n){
    int *p;
    for(p=&array[0];
      p<&array[n];
      p++)
      *p=0;
}

array -- R0, n -- R1
p -- R2, zero -- R3
arraySize -- R12
```

```
MOV p, array
MOV zero, #0
loop2:  STR zero,[p],#4
ADD arraySize,array,n,LSL #2
CMP p,arraySize
BLT loop2
```

Arrays vs. Pointer

```
clear1:

MOV i, 0
MOV zero, 0
loop1:
STR zero,[array,i,LSL #2]
ADD i, i, #1
CMP i, n
BLT loop1
```

```
clear2:

MOV p, array
MOV zero, #0
ADD arraySize,array,n,LSL #2
loop2:
STR zero,[p],#4
CMP p,arraySize
BLT loop2
```

Synchronization

- Two processes access the same memory location

Synchronization

- Two processes access the same memory location
- Results depends on order of execution

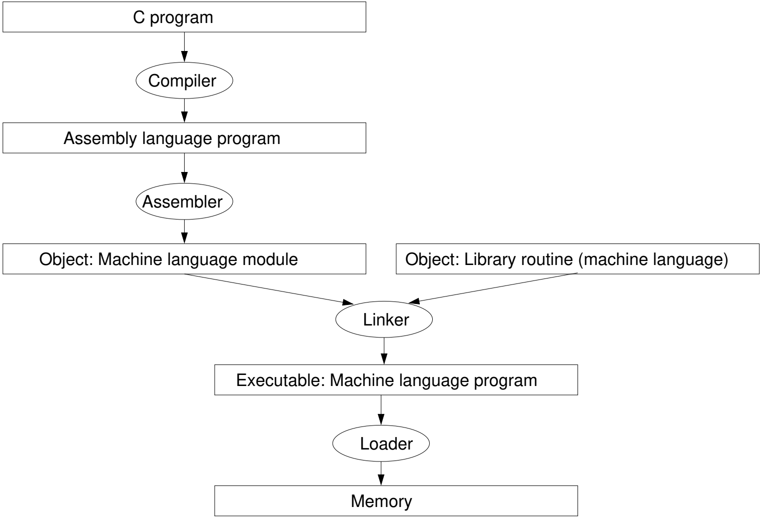
Synchronization

- Two processes access the same memory location
- Results depends on order of execution
- Hardware support is required

Synchronization

- Two processes access the same memory location
- Results depends on order of execution
- Hardware support is required
  - Mutual exclusion
  - Atomic read/write
  - Critical block
  - `SWP` – Atomic exchange

Translation hierarchy for C



Pseudoinstruction

- Most assembler instructions represent machine instructions one-to-one
- A common variation of assembly language instructions often treated as if it were an instruction its own right
- `LDR r0, #constant`
- The assembler determines which instructions to use to create the constant in the most efficient way.

Assembler

- Object file header – Describe the size and position of the other pieces of the object file
- Text segment – Contains the machine language code
- Static data segment – Contains data allocated for the life time of the program
- Relocation information – Identifies instruction and data word that depend on absolute address when the program is loaded into memory.
- Symbol table – Contains the remaining labels that are not defined
- Debugging information – Extra information so as to associate C source file to machine instruction

Linker

- Place code and data module symbolically in memory
- Determine the address of data and instruction labels
- Patch both the internal and external references

Example: object file

Object file header			
	Name	Procedure A	
	Text size	100	
	Data size	20	
Text segment	Address	Instruction	
	0	LDR r0, 0(r3)	
	4	BL 0	
Data segment	0	(X)	
Relocation information	Address	Instruction type	Dependency
	0	LDR	X
	4	BL	B
Symbol table	Label	Address	
	X	–	
	B	–	

Example

Object file header				Executable file header		
	Name	Procedure A			Text size	300
	Text size	100			Data size	50
	Data size	20		Text segment	Address	Instruction
Text segment	Address	Instruction			1000	LDR r0, -6000(r3)
	0	LDR r0, 0(r3)			1004	BL 92
	4	BL 0				
Data segment	0	(X)			1100	STR r1, 6020(r3)
Relocation information	Address	Instruction type	Dependency		1104	BL -108
	0	LDR	X			
	4	BL	B	Data segment	Address	
Symbol table	Label	Address			2000	(X)
	X	–			...	...
	B	–			2020	(Y)
Object file header						
	Name	Procedure B				
	Text size	200				
	Data size	30				
Text segment	Address	Instruction				
	0	STR r1, 0(r3)				
	4	BL 0				
Data segment	0	(Y)				
Relocation information	Address	Instruction type	Dependency			
	0	STR	Y			
	4	BL	A			
Symbol table	Label	Address				
	Y	–				
	A	–				

Loader

- Reads the executable file header to determine size of text and data segment
- Creates an address space large enough to store text and data
- Copies the instructions and data from the executable file into memory
- Copies the parameters (if any) to the main program onto stack
- Initialize the machine registers and sets the stack pointer to first free location
- Jumps to start-up routine that copies the parameters into the argument registers and calls the main routine.

Sort

```
void swap(int v[], int k) {
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}

void sort(int v[], int n){
   int i, j;
   for(i=0; i<n; i++){
     for(j=i-1; j>=0 && v[j]>v[j+1]; j--){
       swap(v, j);
     }
   }
}
```