ECSE324 : Computer Organization

Software — Assemblers, Linkers, Compilers & Debugger

Christophe Dubach
Fall 2021

Revision history:
Christophe Dubach – F2021
Some material from Hamacher, Vranesic, Zaky, and Manjikian, *Computer Organization and Embedded Systems, 6 th ed*, 2012, McGraw Hill and Patterson and Hennessy, *Computer Organization and Design, ARM Edition*, Morgan Kaufmann, 2017,

Timestamp: 2021/10/01 10:03:00

Disclaimer

Lectures are recorded live and posted unedited on *MyCourses* on the same day.

It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the textbook, or ask on the online forum for clarification.

Overview

The big picture

C (file.c)

```c
int i = 1;
int result = 0;
if (n>0) {
  do {
    result += i;
    i++;
  } while (i<=n)
}
```

Compiler →

Assembly (file.asm)

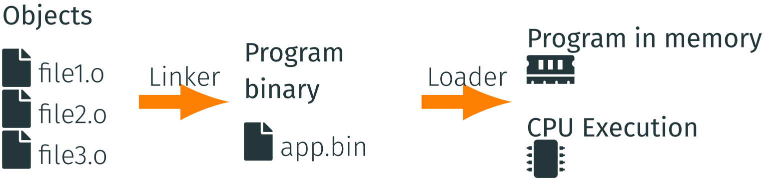
```
n: .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // i
 MOV R2, #0       // result
 CMP R0, #0
 BLE END          // if n<=0

LOOP:
 ADD R2, R2, R1 // result+=i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP         // if i<=n

END:
```

Assembler →

Object (file.o)

Address	Content
00000000	00000003
00000004	e51f000c
00000008	e3a01001
0000000c	e3a02000
00000010	e3500000
00000014	da000003
00000018	e0822001
0000001c	e2811001
00000020	e1510000
00000024	dafffffb
00000028	...

- The compiler turns high-level source code into assembly.
- The assembly code is turned into an object file by the assembler.
- This step is repeated for each input source code that makes up the program.



- Separate object files are linked together to create the program binary.
- When the operating system triggers the program’s execution, the program is placed in memory by the loader and execution starts on the CPU.

Assembler

The assembler is a program that takes assembly source code as input and procudes a binary object file.

Assembly (file.asm)

```
n:  .word 3
 LDR R0, n        // num iter
 MOV R1, #1       // i
 MOV R2, #0       // result
 CMP R0, #0
 BLE END          // if n<=0
LOOP:
 ADD R2, R2, R1 // result+=i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP         // if i<=n
END:
```

Assembler →

Object (file.o)

Address	Content
00000000	00000003
00000004	e51f000c
00000008	e3a01001
0000000c	e3a02000
00000010	e3500000
00000014	da000003
00000018	e0822001
0000001c	e2811001
00000020	e1510000
00000024	dafffffb
00000028	...

This is done by:

- Recognizing directives and instructions, or throw errors;
- Converting instructions to *binary* machine code;
- Laying out instructions in memory;
- Building a symbol table.

Assembling step by step

The assembler proceed by reading the input assembly code line by line. This results in two outputs:

- an Object Program Memory Map
    - stores the binary data/code for each address
- a Symbol Table
    - stores the value associated with each label

Let's see an example of this process.

Object Program Memory Map

Address	Content	Disassembled

Assembly Program

```
n:  .word 3
```

Symbol Table

Name	Value

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	

Assembly Program

```
n: .word 3
```

Symbol Table

Name	Value
n	00000000

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	

Assembly Program

```
n: .word 3

 LDR R0, n       // num iter
```

Symbol Table

Name	Value
n	00000000

How do we know what to do with `LDR R0, n` ?

Let's check the manual:

A8.8.64 LDR (immediate, ARM)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-292.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>, [<Rn>{, #+/-<imm12>}]
LDR<c> <Rt>, [<Rn>], #+/-<imm12>
LDR<c> <Rt>, [<Rn>, #+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	0	W	1	Rn	Rt	imm12

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-214.

```
if Rn == '1111' then SEE LDR (literal);
if P == '0' && W == '1' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '0' && imm12 == '000000000100' then SEE POP;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

source: ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]

Assembly Program

```
n:  .word 3

 LDR R0, n       // num iter
```

Symbol Table

Name	Value
n	00000000

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]

Assembly Program

```
n:  .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
```

Symbol Table

Name	Value
n	00000000

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1

Assembly Program

```
n: .word 3

 LDR R0, n       // num iter
 MOV R1, #1      // iter var i
```

Symbol Table

Name	Value
n	00000000

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1

Assembly Program

```
n: .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
 MOV R2, #0       // result
```

Symbol Table

Name	Value
n	00000000

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0

Assembly Program

```
n: .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
 MOV R2, #0       // result
```

Symbol Table

Name	Value
n	00000000

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0

Assembly Program

```
n: .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
 MOV R2, #0       // result
 CMP R0, #0
```

Symbol Table

Name	Value
n	00000000

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0

Assembly Program

```
n: .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
 MOV R2, #0       // result
 CMP R0, #0
```

Symbol Table

Name	Value
n	00000000

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0

Assembly Program

```
n: .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
 MOV R2, #0       // result
 CMP R0, #0
 BLE END          // if n<=0
```

Symbol Table

Name	Value
n	00000000

We have a problem when we encounter a *forward reference*.

What is the value of END?

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??

Assembly Program

```
n:  .word 3

 LDR R0, n          // num iter
 MOV R1, #1         // iter var i
 MOV R2, #0         // result
 CMP R0, #0
 BLE END            // if n<=0
```

Symbol Table

Name	Value
n	00000000
END	????????

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??

Assembly Program

```
n:  .word 3

 LDR R0, n          // num iter
 MOV R1, #1         // iter var i
 MOV R2, #0         // result
 CMP R0, #0
 BLE END            // if n<=0

LOOP:
```

Symbol Table

Name	Value
n	00000000
END	????????

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??

Assembly Program

```
n:  .word 3

 LDR R0, n         // num iter
 MOV R1, #1        // iter var i
 MOV R2, #0        // result
 CMP R0, #0
 BLE END           // if n<=0

LOOP:
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??

Assembly Program

```
n: .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
 MOV R2, #0       // result
 CMP R0, #0
 BLE END          // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1

Assembly Program

```
n:  .word 3

 LDR R0, n         // num iter
 MOV R1, #1        // iter var i
 MOV R2, #0        // result
 CMP R0, #0
 BLE END           // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1

Assembly Program

```
n:  .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
 MOV R2, #0       // result
 CMP R0, #0
 BLE END          // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1

Assembly Program

```
n:  .word 3

 LDR R0, n         // num iter
 MOV R1, #1        // iter var i
 MOV R2, #0        // result
 CMP R0, #0
 BLE END           // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1

Assembly Program

```
n:  .word 3

 LDR R0, n          // num iter
 MOV R1, #1         // iter var i
 MOV R2, #0         // result
 CMP R0, #0
 BLE END            // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
 CMP R1, R0
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1
00000020	e1510000	cmp r1, r0

Assembly Program

```
n:  .word 3

 LDR R0, n          // num iter
 MOV R1, #1         // iter var i
 MOV R2, #0         // result
 CMP R0, #0
 BLE END            // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
 CMP R1, R0
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1
00000020	e1510000	cmp r1, r0

Assembly Program

```
n:  .word 3

 LDR R0, n          // num iter
 MOV R1, #1         // iter var i
 MOV R2, #0         // result
 CMP R0, #0
 BLE END            // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP           // if i<=n
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

How to encode this BLE instruction? Let’s check the manual:

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
B<c> <label>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	0	imm24																							

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-214.

```
imm32 = SignExtend(imm24:'00', 32);
```

<label> The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset.

Permitted offsets are:

Encoding T1	Even numbers in the range –256 to 254
Encoding T2	Even numbers in the range –2048 to 2046
Encoding T3	Even numbers in the range –1048576 to 1048574
Encoding T4	Even numbers in the range –16777216 to 16777214
Encoding A1	Multiples of 4 in the range –33554432 to 33554428.

source: ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition

Target is encoded as an offset from the PC in Mutiples of 4 bytes.

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1
00000020	e1510000	cmp r1, r0
00000024	daffffb	ble 0x18

Assembly Program

```
n:  .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // iter var i
 MOV R2, #0       // result
 CMP R0, #0
 BLE END          // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP         // if i<=n
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1
00000020	e1510000	cmp r1, r0
00000024	dafffffb	ble 0x18

Assembly Program

```
n:  .word 3

 LDR R0, n         // num iter
 MOV R1, #1        // iter var i
 MOV R2, #0        // result
 CMP R0, #0
 BLE END           // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP          // if i<=n

END:
```

Symbol Table

Name	Value
n	00000000
END	????????
LOOP	00000018

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1
00000020	e1510000	cmp r1, r0
00000024	daffffb	ble 0x18
00000028	...	...

Assembly Program

```
n:  .word 3

 LDR R0, n         // num iter
 MOV R1, #1        // iter var i
 MOV R2, #0        // result
 CMP R0, #0
 BLE END           // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP          // if i<=n

END:
```

Symbol Table

Name	Value
n	00000000
END	00000028
LOOP	00000018

`END` is only known towards the end of the process.

$\Rightarrow$

Need to run a second pass through the instructions to patch up.

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da??????	ble ??END??
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1
00000020	e1510000	cmp r1, r0
00000024	dafffffb	ble 0x18
00000028	...	...

Symbol Table

Name	Value
n	00000000
END	00000028
LOOP	00000018

Assembly Program

```
n:  .word 3

 LDR R0, n          // num iter
 MOV R1, #1         // iter var i
 MOV R2, #0         // result
 CMP R0, #0
 BLE END            // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP           // if i<=n

END:
```

Object Program Memory Map

Address	Content	Disassembled
00000000	00000003	
00000004	e51f000c	ldr r0, [pc, #-12]
00000008	e3a01001	mov r1, #1
0000000c	e3a02000	mov r2, #0
00000010	e3500000	cmp r0, #0
00000014	da000003	ble 0x28
00000018	e0822001	add r2, r2, r1
0000001c	e2811001	add r1, r1, #1
00000020	e1510000	cmp r1, r0
00000024	dafffffb	ble 0x18
00000028	...	...

Assembly Program

```
n:  .word 3

 LDR R0, n          // num iter
 MOV R1, #1         // iter var i
 MOV R2, #0         // result
 CMP R0, #0
 BLE END            // if n<=0

LOOP:
 ADD R2, R2, R1 // result += i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP           // if i<=n

END:
```

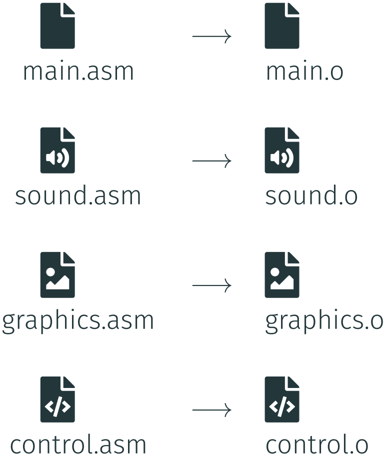
Symbol Table

Name	Value
n	00000000
END	00000028
LOOP	00000018

Linker

Progams are generally written using multiple files to increase modularity and code reusage.

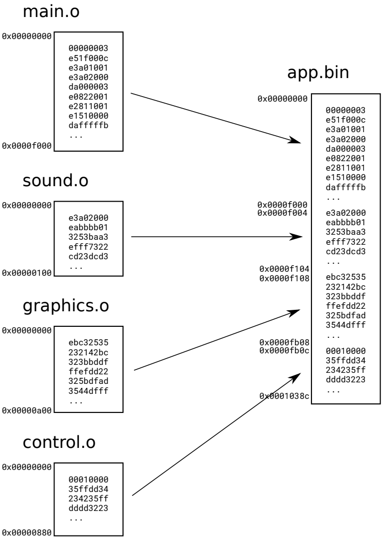
Each file is assembled separately.



How do we create a single binary for the entire application?

Linker

The linker concatanates different object files into a single binary.



What are the potential problems?

What are the potential problems?

- Data and instructions might end up in different locations (addresses) than they were in the object files.
  - ✔ Not a big issue for PC-relative addresses (*e.g.*, Load/Branch)
  - ⚠ If absolute address are used, will need to *relocate* them

What are the potential problems?

- Data and instructions might end up in different locations (addresses) than they were in the object files.
  - ✔ Not a big issue for PC-relative addresses (*e.g.*, Load/Branch)
  - ⚠ If absolute address are used, will need to *relocate* them
- When calling a function from another file, its symbol is unknown at assembly time.
  - ⚠ the linker will have to deal with that using an list of external symbols

Example

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
 ADD A1, A1, #1
 CMP A1, V1
 BLE LOOP
```

otherfile.asm

```
externalFun:
 ADD A1, A1, A2
```

When assembling file `main.asm`, we have an unknown symbol: `externalFun`

Solution:

Keep track of unknown external symbols during assembly, and let the linker patch up later.

Assembling step

Address	Content	Disassembled
...	...	...

main.asm

```
...
LOOP:
```

Symbol Table

Name	Value

External Symbols

Name	Value

Assembling step

Address	Content	Disassembled
...	...	...

main.asm

```
...
LOOP:
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value

Assembling step

Address	Content	Disassembled
...	...	...

main.asm

```
...
LOOP:
 MOV A2, V2
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5

main.asm

```
...
LOOP:
 MOV A2, V2
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value
externalFun	???

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value
externalFun	???

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??
00000014	e49de004	pop lr

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value
externalFun	???

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??
00000014	e49de004	pop lr

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
 ADD A1, A1, #1
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value
externalFun	???

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??
00000014	e49de004	pop lr
00000018	e2800001	add r0, r0, #1

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
 ADD A1, A1, #1
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value
externalFun	???

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??
00000014	e49de004	pop lr
00000018	e2800001	add r0, r0, #1

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
 ADD A1, A1, #1
 CMP A1, V1
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value
externalFun	???

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??
00000014	e49de004	pop lr
00000018	e2800001	add r0, r0, #1
0000001c	e1510004	cmp r0, r4

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
 ADD A1, A1, #1
 CMP A1, V1
```

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value
externalFun	???

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??
00000014	e49de004	pop lr
00000018	e2800001	add r0, r0, #1
0000001c	e1510004	cmp r0, r4

Symbol Table

Name	Value
LOOP	0x00000008

External Symbols

Name	Value
externalFun	???

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
 ADD A1, A1, #1
 CMP A1, V1
 BLE LOOP
```

Assembling step

Address	Content	Disassembled
...	...	...
00000008	e1a01005	mov r1, r5
0000000c	e52de004	push lr
00000010	eb??????	BL ??
00000014	e49de004	pop lr
00000018	e2800001	add r0, r0, #1
0000001c	e1510004	cmp r0, r4
00000020	daffff8	ble 0x08
...	...	...

Symbol Table

Name	Value
LOOP	0x00000008

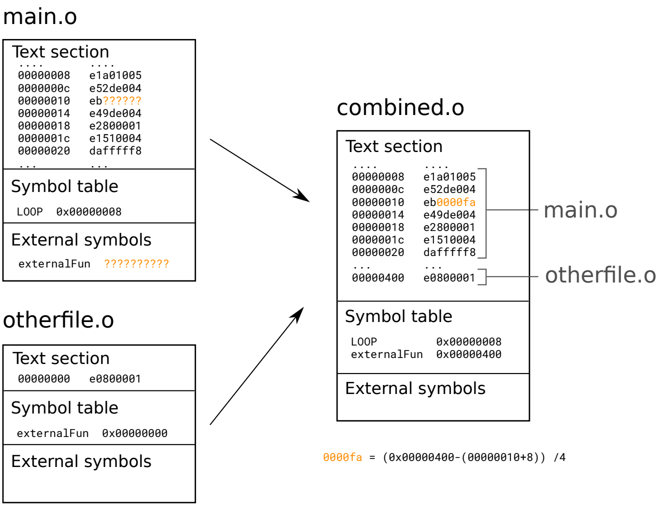
External Symbols

Name	Value
externalFun	???

main.asm

```
...
LOOP:
 MOV A2, V2
 PUSH {LR}
 BL externalFun
 POP {LR}
 ADD A1, A1, #1
 CMP A1, V1
 BLE LOOP
```

Linking step



Demo

Last but not least, to make an actual program, we need a *start* symbol.

This start symbol specifies the address of the first instruction that should execute when the program starts.

As seen in the labs, it is declared in ARM assembly like this:

```
.global _start
_start:
        first insruction goes here
```

When coming from C, this start symbol will be set so that the `main` function will be the first user-written function to execute.

Libraries

It is possible to reuse the same object files across multiple programs.

If the linking is done at runtime, we are talking about a library:

- Shared object on Unix (`.so`)
- Dynamic Link Library on Windows (`.dll`)

Advantages of shared libraries:

- if a library is updated (*e.g.*, bug fix), all programs benefit.
- Using operating system mechanisms (*i.e.*, virtual memory, copy-on-write), the library code can be shared in memory, reducing memory consumption.

Loader

What happens when you want to run a program?

The operating system uses the **loader**, a specialized program, to start the user program.

Loader

The loader performs mainly four steps:

1. Allocate space in memory for the program to be loaded;
2. *Load* the program into the allocated memory from file;
3. Set the PC to the start of the program code;
4. Free up allocated memory once the program has finished.

Disk

Memory

CPU

PC

prog.bin

Text section

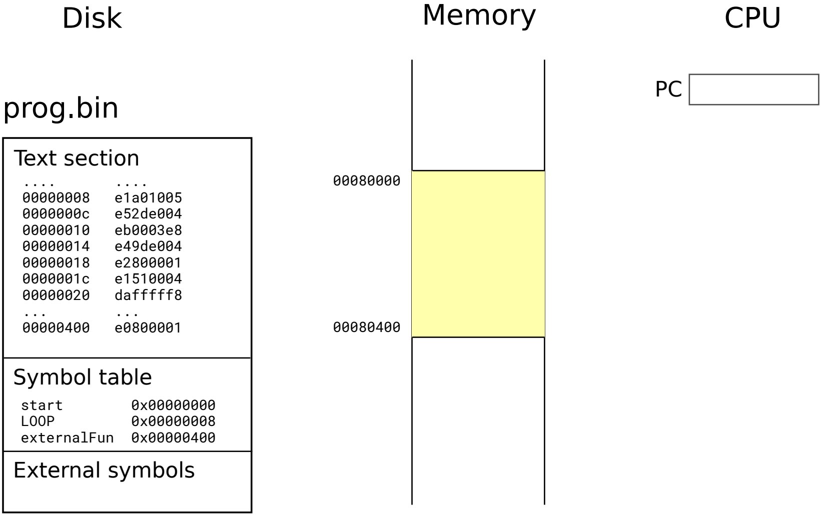
```
....        ....
00000008    e1a01005
0000000c    e52de004
00000010    eb0003e8
00000014    e49de004
00000018    e2800001
0000001c    e1510004
00000020    dafffff8
...         ...
00000400    e0800001
```

Symbol table

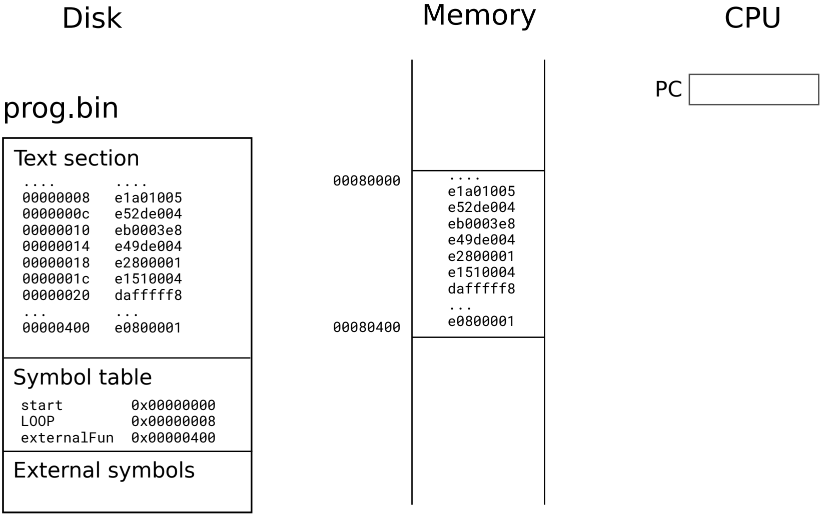
```
start        0x00000000
LOOP         0x00000008
externalFun  0x00000400
```

External symbols

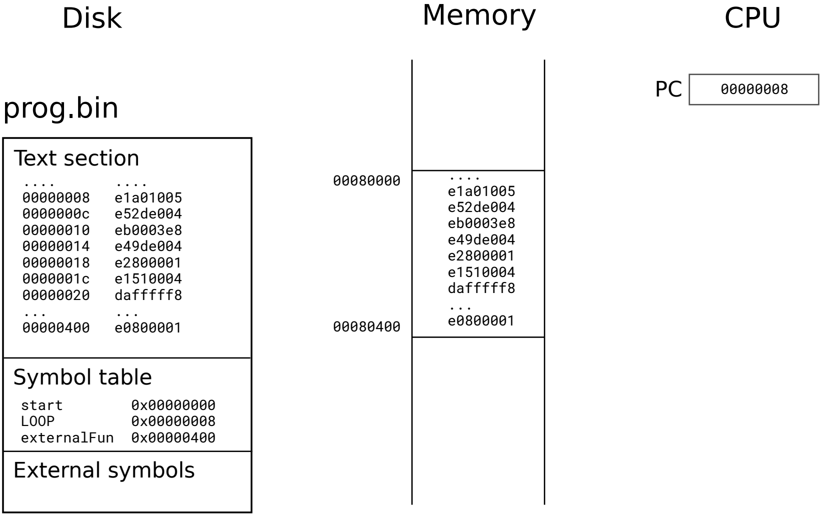
1. Space allocation



2. Program loading



3. Program execution



4. Deallocation

Disk | Memory | CPU

PC

prog.bin

Text section

```
....        ....
00000008   e1a01005
0000000c   e52de004
00000010   eb0003e8
00000014   e49de004
00000018   e2800001
0000001c   e1510004
00000020   dafffff8
...         ...
00000400   e0800001
```

Symbol table

```
start         0x00000000
LOOP          0x00000008
externalFun   0x00000400
```

External symbols

Compiler

The Compiler

C (file.c)

```c
int i = 1;
int result = 0;
if (n>0) {
  do {
    result += i;
    i++;
  } while (i<=n)
}
```

Compiler

Assembly (file.asm)

```
n:  .word 3

 LDR R0, n        // num iter
 MOV R1, #1       // i
 MOV R2, #0       // result
 CMP R0, #0
 BLE END          // if n<=0

LOOP:
 ADD R2, R2, R1 // result+=i
 ADD R1, R1, #1 // i++
 CMP R1, R0
 BLE LOOP         // if i<=n

END:
```

- The compiler’s main job consists of converting a program written in a high-level language (*e.g.*, C, Java) into assembly instructions.
- Usually, a compiler also invokes the assembler to generate the object files and the linker to produce the final program binary.

Compilation is complex and is the topic of an entire course: COMP-520.

This involves:

- Detecting syntax, semantic and typing errors;
- Building internal data structures to represent and manipulate the progam;
- Applying optimizations (*e.g.*, dead code elimination, constant propagation);
- Allocating registers for each operations performed;
- And finally, producing the assembly instructions.

As part of this process, the compiler deals with tedious tasks such as stack management and subroutine calls.

The first “compiler”, the A-0 System, was in fact designed for that purpose by Grace Hopper in 1952!

Grace Hopper, US Navy

Debugger

Imagine:

- you have implemented your algorithm
- you have dealt with compiler errors and warnings
- you have dealt with the linking errors
- but your code still does not work:
    - incorrect output
    - crash
    - never stops
    - …

What do you do? — no, you don't give up yet!

Debugging strategies

- Use print statement in your code to track:
  - iteration variable, condition expressions, ...
- Use assertions to check your assumptions
- Unit tests
- Rubber duck debugging

Or, you could use a debugger!

The Debugger

The debugger is a tool that you can use to observe your program while it is running.

- You can observe the value of any variables or memory location.
- You can pause the execution of the program and resume it.
- You can inspect the call stack.
- You can even modify the content of any variables or memory location!

Most development toolchains and modern IDEs come with a debugger.

To support these features, a debugger relies on:

- debugging information stored in the object file:
  - mapping between addresses of instruction and origin (*e.g.*, filename, variable name or function name)
- control over program execution exposed by the operating system and hardware.

When compiled with debugging information enabled, the produced object files / binary will be larger and execution will be slower.

Demo

The ability to control a program execution through a debugger usually requires special support in the operating system and hardware.

In general this is achieved through hardware interrupts that gives back control to the operating system under certain conditions such as:

- after every instruction using *trace mode*;
- after reaching a specific address; a *breakpoint*.

In such case, when the interrupt routine executes (more on this later in the course), the control will pass onto the debugger. The user can now run debugger commands to:

- inspect and modify memory or registers;

Then, control returns normally to the application until the next interrupt.

Operating System

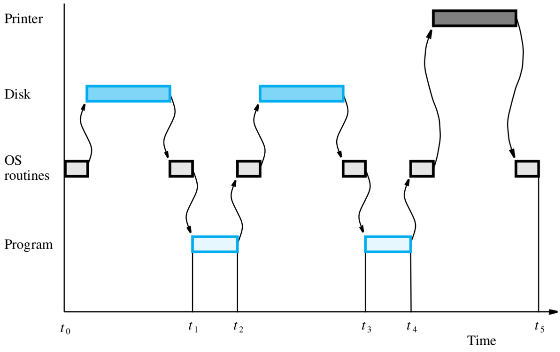
The Operating System (OS)

The OS is responsible for managing the resources of the machine efficiently. In particular, it:

- loads program (the loader is part of the OS);
- coordinates execution of different applications, providing the illusion of parallel execution:
  - when an application is waiting for a long I/O operation to finish, it might schedule another one instead;
- ensures applications cannot interfer with one another in harmful ways;
- manages memory allocation and I/O requests.

Access to I/O devices from the user application is done through the OS (unless there is no OS).

Example of application running:

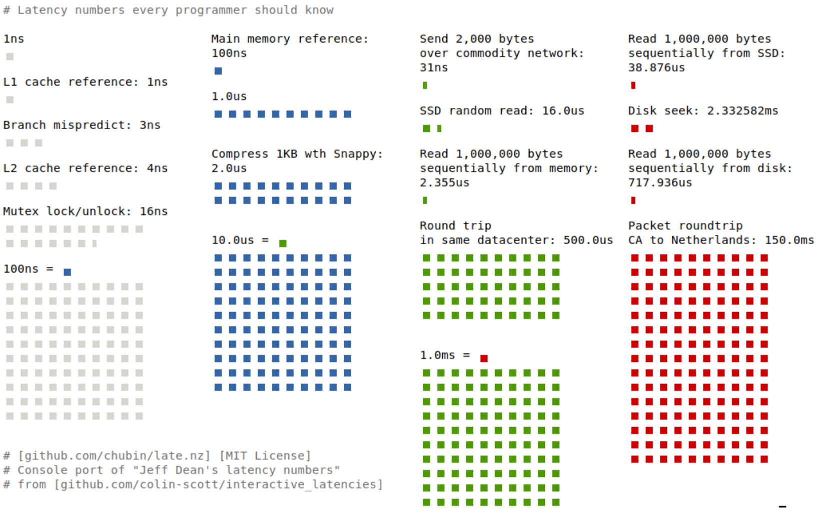


In this example, system resources are under-utilized:

- CPU is sometimes waiting for I/O devices;
- I/O devices are sometimes idle.

A typicaly OS would run multiple applications concurrently to best utilize the resources available and maximize overall *throughput*.

Latency Numbers Every Programmer Should Know (2020)



Conclusions

This lecture has introduced the typical software stack that helps to operate computers. We have seen that it is composed of:

- the assembler, linker and loader;
- the compiler;
- the debugger;
- and the operating system;

The next lecture will look at:

- how to interact with I/O devices from software (assembly);
- the hardware interface of I/O devices.