MT Review Session: Number Rep, FP

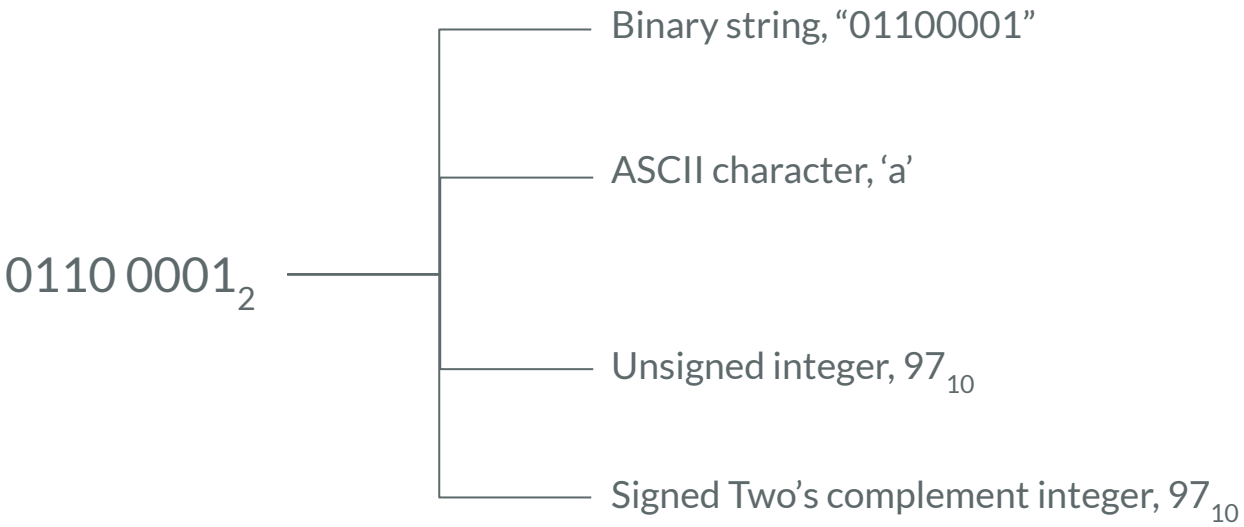
TA: Tyler Yang
Slide deck credit: Eran Kohen Behar, Kim Pham

Logistics

- 11-12 AM: Number Rep, Floating Point, CALL
- 12-1 PM: C Review

Number Representation

Everything is bits!!



Binary Representation Tricks

How can we represent powers of 2?

- Since each bit represents a power of two, only one bit is 1, the rest are 0.
- e.g.: $32 = 2^5 = 0b\ \underset{2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0}{1\ 0\ 0\ 0\ 0\ 0}$

How can we represent number less than a power of two?

- All 1s up to but not including the bit for the next power of two.
- Think of how you would subtract 1 from the number above
- e.g.: $31 = 2^5\text{-}1 = 0b\ \underset{2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0}{0\ 1\ 1\ 1\ 1\ 1}$

Representing Negative Numbers

Sign-Magnitude Notation: Most significant bit represents sign, the rest represents magnitude.

- e.g. -17_{10} => 0b $\underbrace{1\ 1\ 0\ 0\ 0\ 1}_{-1 \times 2^4 + \ldots + 2^0}$

Representing Negative Numbers but Better

Two's Complement: This is the standard!

- Most significant bit is negative instead of positive!
- e.g. -17_{10} => 0b $\underline{1\ 0\ 1\ 1\ 1\ 1}$ (assume 6 bit integers)

  $-2^5\ \ 2^4\ \ 2^3\ \ 2^2\ \ 2^1\ \ 2^0 = -32 + 8 + 4 + 2 + 1 = 15 - 32 = -17$
-
- But this is way too difficult and annoying to come up with, so we don't :)
- Instead, find the positive number, flip the bits, and add 1:
- e.g. 17_{10} => 0b 0 1 0 0 0 1
- when we flip the bits,
- 0b 1 0 1 1 1 0 . Then add 1:
- -17_{10} => 0b 1 0 1 1 1 1

The most significant bit still represents the sign of the number!

Still Representing Negative Numbers

Bias Notation:

- Okay now we simplify things. Treat the bits as unsigned, then add a bias.
- e.g. 8-bit integer with bias of -252:
  - 0b 0000 0000 + $(-252)_{10}$ = 0 - 252 = -252_{10}
  - 0b 1111 1111 + $(-252)_{10}$ = 255 - 252 = 3_{10}
- The bias can be negative, positive, or 0! Bias of 0 is the same as regular unsigned.
- How to convert?
- e.g. -17 with 8 bits and a bias of -31:
  - x + (-31) = -17 → x = 14 → represent 14 in 8-bit unsigned!
  - 14_{10} = 0b 0000 1110
  - Now, 0b 0000 1110 + $(-31)_{10}$ = 14 - 31 = -17_{10}

Hexadecimals

Think of hex as a more compact version of binary, nothing more!

Remember, every 4 bits of binary correspond to 1 hex digit.

Fall 2019 Question 1

Negate the following nibble binary/hex numbers, or write N/A if not possible. Remember to write your answer in the appropriate base.

Representation	Binary/Hex	Negation
Unsigned	0b0101	
Bias = -7	0b0100	
Bias = -7	0xF	
Two's Complement	0b1100	
Two's Complement	0xA	

Fall 2019 Question 1

Negate the following nibble binary/hex numbers, or write N/A if not possible. Remember to write your answer in the appropriate base.

Representation	Binary/Hex	Negation
Unsigned	0b0101	N/A (no negatives in unsigned)
Bias = -7	0b0100	0b1010
Bias = -7	0xF	N/A (out of range)
Two's Complement	0b1100	0b0100
Two's Complement	0xA	0x6

Cheat Sheet Tips and Tricks

N bits represent 2^N “things”

- N-bit unsigned: $[0, 2^N - 1]$
  - smallest: all 0s
  - largest: all 1s
- N-bit Two’s Complement: $[-2^{N-1}, 2^{N-1} - 1]$
  - smallest: 1 in sign bit, followed by all 0s
  - largest: 0 in sign bit, followed by all 1s
  - -1_{10} represented by all 1s!
  - to negate, flip bits and add 1
  - ^^ Be sure to understand why all of these are true!
- N-bit bias notation with bias b: $[b, 2^N - 1 + b]$
  - smallest: all 0s
  - largest: all 1s

Summer 2022 Question 1.3

Q1.3 (3 points) Which of the following representations have more than one representation of 0? Select all that apply.

- ☐ (A) Two's complement
- ☐ (B) Sign-magnitude
- ☐ (C) An IEEE-754 standard double-precision float
- ☐ (D) Bias notation
- ☐ (E) None of the above

Summer 2022 Question 1.3

Q1.3 (3 points) Which of the following representations have more than one representation of 0? Select all that apply.

- ☐ (A) Two's complement
- ☐ (B) Sign-magnitude
- ☐ (C) An IEEE-754 standard double-precision float
- ☐ (D) Bias notation
- ☐ (E) None of the above

Floating Point

Floating Point Representation

The below table shows the bit breakdown for the single precision (32-bit) representation. The leftmost bit is the MSB and the rightmost bit is the LSB.

1	8	23
Sign	Exponent	Mantissa/Significand/Fraction

For normalized floats:

Value $= (-1)^{Sign} * 2^{Exp+Bias} * 1.\textbf{significand}_2$

For denormalized floats:

Value $= (-1)^{Sign} * 2^{Exp+Bias+1} * 0.\textbf{significand}_2$

Exponent	Significand	Meaning
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

- Sign bit determines +/-
- Exponent is biased by -(2^{n-1} - 1), -127 for 32-bit floats
- Mantissa represents the bits after the decimal point.
- Single bit before the decimal point is usually 1, but 0 for denorm numbers!

$(-1) \times 1.0110101_2 \times 2^{100 + (-127)}$

Sign bit

mantissa

exponent

replace this with a 0. if the exponent is 0

bias. add 1 if exponent is 0.

Spring 2019 MT2 Question 4

Consider a modified floating point scheme where we opt to use 7 bits for the exponent and 24 bits for the significand but is otherwise the same as IEEE 754 Single Precision Floating Point.

b) Represent 18.75 in our new Floating Point Scheme.

c) What is the largest finite value you can represent?

Spring 2019 MT2 Question 4)a

Consider a modified floating point scheme where we opt to use 7 bits for the exponent and 24 bits for the significand but is otherwise the same as IEEE 754 Single Precision Floating Point.

a) What is the new bias for our Floating Point Scheme?

Spring 2019 MT2 Question 4)a

Consider a modified floating point scheme where we opt to use 7 bits for the exponent and 24 bits for the significand but is otherwise the same as IEEE 754 Single Precision Floating Point.

a) What is the new bias for our Floating Point Scheme?

We want the bias to center the exponent around 0 (equal number of positive and negative numbers)

In IEEE-754, 8 bits of exponent have a bias of -127. Range is [-127, 128]

For a 7-bit exponent (unsigned range [0, 127]), we can use a bias of -63 to shift the range to [-63, 64]

In general, for n-bit numbers, a bias of $2^{n-1}-1$ will center the range around 0.

Spring 2019 MT2 Question 4)b

Consider a modified floating point scheme where we opt to use 7 bits for the exponent and 24 bits for the significand but is otherwise the same as IEEE 754 Single Precision Floating Point.

b) Represent 18.75 in our new Floating Point Scheme.

Spring 2019 MT2 Question 4)b

Consider a modified floating point scheme where we opt to use 7 bits for the exponent and 24 bits for the significand but is otherwise the same as IEEE 754 Single Precision Floating Point.

b) Represent 18.75 in our new Floating Point Scheme.

1. Represent the number in binary.
2. Normalize so that you have 1 digit before the decimal point. For denorms, you will need to shift it so that a 0 is before the decimal.
3. Calculate exponent bits
4. Put it all together

1. 18.75_{10} = 0b 10010.11
2. 10010.11 = $1.001011 \times 2^4 \rightarrow$ mantissa = 001011000…
3. exp + bias = 4 → exp = 67 → 0b 1000011
4. SEEEEEEEMMM… → 0 1000011 0010110000000…
5. → 0x432C0000

Spring 2019 MT2 Question 4)c

Consider a modified floating point scheme where we opt to use 7 bits for the exponent and 24 bits for the significand but is otherwise the same as IEEE 754 Single Precision Floating Point.

c) What is the largest finite value you can represent?

Spring 2019 MT2 Question 4)c

Consider a modified floating point scheme where we opt to use 7 bits for the exponent and 24 bits for the significand but is otherwise the same as IEEE 754 Single Precision Floating Point.

c) What is the largest finite value you can represent?

If it were unsigned, we would make everything all 1s. But in floating points, a mantissa of all 1s corresponds to infinity or NaN. So let's just make the mantissa 1 less than that.

S = 0 (positive)
EEEEEEE = $1111110_2 = 126_{10}$
1.MMM...MMM = 1.111...111 = $1 + (2^{24} - 1) * 2^{-24} = 2 - 2^{-24}$

Putting it all together:

$1.MM...MM * 2^{EEEEEEE + bias} = (2 - 2^{-24}) * 2^{126 - 63}$
$= 2^{64} - 2^{39}$

Spring 2022 MT Question 2

For the following three subparts, assume that we are working with a binary floating point representation, which follows IEEE-754 standard conventions, but which has 3 exponent bits (and a standard exponent bias of -3) and 4 significand bits.

Q2.3 (3 points) Convert -12 to its floating point representation under this floating point system. Express your answer in binary, including the relevant prefix.

Spring 2022 MT Question 2

For the following three subparts, assume that we are working with a binary floating point representation, which follows IEEE-754 standard conventions, but which has 3 exponent bits (and a standard exponent bias of -3) and 4 significand bits.

Q2.3 (3 points) Convert -12 to its floating point representation under this floating point system. Express your answer in binary, including the relevant prefix.

Solution:

$12 = \texttt{0b1100} = \texttt{0b}1.1000 \times 2^3$

Significand = $\texttt{0b}1000$

Exponent = $3 - (-3) = 6 = \texttt{0b}110$

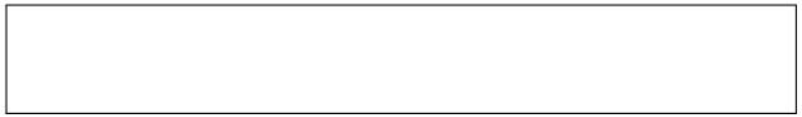
Sign bit = 1 (negative)

$\texttt{0b}11101000 \longrightarrow \texttt{0b}11101000$

Spring 2022 MT Question 2

For the following three subparts, assume that we are working with a binary floating point representation, which follows IEEE-754 standard conventions, but which has 3 exponent bits (and a standard exponent bias of -3) and 4 significand bits.

Q2.5 (3 points) What is the smallest positive number that can be represented by this system? Express your answer as an odd integer multiplied by a power of 2.



Spring 2022 MT Question 2

For the following three subparts, assume that we are working with a binary floating point representation, which follows IEEE-754 standard conventions, but which has 3 exponent bits (and a standard exponent bias of -3) and 4 significand bits.

Q2.5 (3 points) What is the smallest positive number that can be represented by this system? Express your answer as an odd integer multiplied by a power of 2.

Solution:

Smallest significand = `0b0001`

Smallest exponent = `0b000` $= 0 + (-3) + 1 = -2$

$0.0001 \times 2^{-2} = 1 \times 2^{-6}$

Spring 2023 MT Question 3.4

For this question, assume that we are working with a binary floating point representation, which follows IEEE-754 standard conventions, but has 5 exponent bits (and a standard bias of -15) and 10 mantissa bits.

Q3.4 (5 points) How many non-zero numbers **x** are there in this floating point system where **x** and **2x** differ by exactly 1 bit?

Write your answer as a sum or difference of unique powers of 2 (e.g. $2^3 - 2^2 + 2^1$).

How do we multiply by 2?

Spring 2023 MT Question 3.4

Q3.4 (5 points) How many non-zero numbers **x** are there in this floating point system where **x** and **2x** differ by exactly 1 bit?

Write your answer as a sum or difference of unique powers of 2 (e.g. $2^3 - 2^2 + 2^1$).

We have to be able to multiply a number without "carrying over bits". (ie. we cannot shift left the mantissa because that would modify more than one bit). This also means that multiplying by 2 of the exponent doesn't work unless the LSB of the exponent is 0 (If the exponent bits are 0b00001, multiplying by 2 requires 0b00010 which changes 2 bits).

Fixing the LSB of the exponent to 0, we now are left with 2^{15} numbers (Reminder we have a 16 bit floating point system!)

However, we have to consider which of these numbers do not work. Any denorm number (if the exponent is 0) doesn't work! (if we change the exponent bits -> we get a normalized number which is never going to be 2 times the denorm number due to the implicit 0 becoming a 1 at the front of the mantissa)

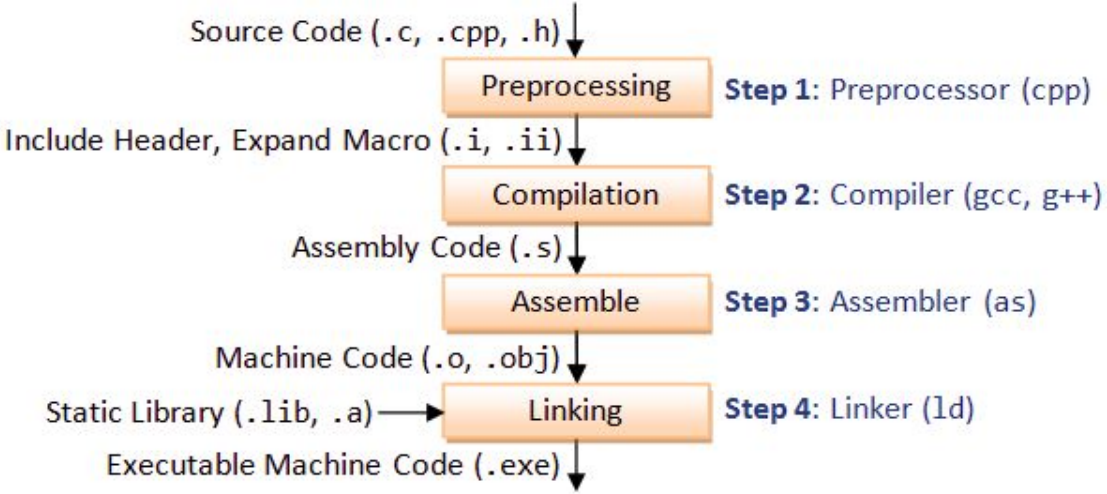
In this scheme, we know there are 2^{11} denorm numbers (Fixing the 5 exponent bits)

Finally, we have to get rid of the infs and NaNs. There are 2^{11} of these numbers. We achieve this by also fixing the 5 exponent bits to all 1s

Therefore our final answer is $2^{15} - 2^{11} - 2^{11} = 2^{15} - 2^{12}$

CALL

CALL Overview



Compilation/Compiler

Assembling/Assembler

Linking/Linker

Loading/Loader

NOTE: preprocessors handle preprocessor directives such as:

- #define statements
- #include statements
- Conditional statements (#if, #endif, #ifdef, #else, etc...)
- Macros (objects/data, functions, etc...)

Output: pure C file

Compiler / Compilation

C or other low level language → RISC-V or other assembly language

This does not happen for high level languages like Python

Optimizations occur at this level (gcc will make your code better and faster)

Output will contain:

Labels, pseudoinstructions, relative addressing

Assembler / Assembly

RISC-V or other assembly language → Object file

Assembler must take two passes over the assembly code (why?)

Generates two tables for future use:

Symbol Table: labels and their relative addresses where they're defined

Relocation table: indicates parts of the code that will need to be calculated and changed later. (external labels, data in static section, etc.)

2-Pass Assembler Example

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing

Relocation Table for File A

Label Name	State

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start

Relocation Table for File A

Label Name	State

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start

Relocation Table for File A

Label Name	State
loop	?

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start

Relocation Table for File A

Label Name	State
loop	?

2-Pass Assembler Example

First Pass

```
file_a.s
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start

Relocation Table for File A

Label Name	State
loop	?
malloc	?

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

Relocation Table for File A

Label Name	State
loop	?
malloc	?

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j loop
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j ~~loop~~ fa_start + 44
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + (4 * 11)

Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

2-Pass Assembler Example

First Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j ~~loop~~ fa_start + 44
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

2-Pass Assembler Example

Second Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, loop
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j ~~loop~~ fa_start + 44
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

Relocation Table for File A

Label Name	State
loop	?
malloc	?
done	?

2-Pass Assembler Example

Second Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, ~~loop~~
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j ~~loop~~ fa_start + 44
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	?

2-Pass Assembler Example

Second Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, ~~loop~~
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, done
       # <4 instructions here>
15     j ~~loop~~ fa_start + 44
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	?

2-Pass Assembler Example

Second Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, ~~loop~~
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, ~~done~~
       # <4 instructions here>
15     j ~~loop~~ fa_start + 44
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	fa_start + 64

2-Pass Assembler Example

Second Pass

file_a.s

```
00 func_name:
       addi sp, sp, -4
01     beq a0, x0, ~~loop~~
       # <8 instructions here>
10     jal ra, malloc
11 loop: bneq t0, x0, ~~done~~
       # <4 instructions here>
15     j ~~loop~~ fa_start + 44
16 done: ret
```

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	fa_start + 64

2-Pass Assembler Example

Second Pass

Q: what is fa_start?

Q: what does it mean that malloc's state in the relocation table is still "?"

Symbol Table for File A

Label Name	Relative Addressing
func_name	fa_start
loop	fa_start + 44
done	fa_start + (4 * 16)

Relocation Table for File A

Label Name	State
loop	fa_start + 44
malloc	?
done	fa_start + 64

2-Pass Assembler Example

Second Pass

Q: what is fa_start?

A: this is the "base address" we're using in this example; the assembly and system determine where we're offsetting from.

Q: what does it mean that malloc's state in the relocation table is still "?"

A: this tells us that malloc wasn't defined in File A; thus it must be defined in a separate user file OR in a separate library. Both of these are considered "external references" and should be resolved in the linking stage.

Symbol Table for File A

Label Name	Relative Addressing
`func_name`	`fa_start`
`loop`	`fa_start + 44`
`done`	`fa_start + (4 * 16)`

Relocation Table for File A

Label Name	State
`loop`	`fa_start + 44`
`malloc`	?
`done`	`fa_start + 64`

.o files

- Object file header
  - Size + position of later components
- Text segment ⇒ machine code
- Data segment ⇒ bin of all static data
- Relocation table
- Symbol table
- Debugging information

Linker / Linking

Multiple object files → .exe file (executable)

Categorizes code segments from object files and puts them together

(the linker decides on the order)

Resolves all references

- References from user tables are resolved using symbol tables
- External references resolved with static/dynamic linking of lib files
- Goes through all relocation table entries

All absolute addresses are filled in!

Q: how does the linker know what the absolute addresses are if we haven't specified a place in memory to load to?

A: a standard location in memory specified as the start of the text of a program is 0x40000000; at this point we still assume one process ever runs at a time so there will not be anything else there.

.exe files

- Executable header
  - Size of text and data segments
- Text + data segments

Loader / Loading

.exe file → Puts the program on memory and gets it ready to run.

Sets up necessary space on memory for text and data.

Initializes stack to hold arguments from the user.

Initializes registers

and more…