CS356: Discussion #13

Linking and Processor Organization

Marco Paolieri (paolieri@usc.edu)
Illustrations from CS:APP3e textbook

USC University of Southern California

Schedule: Exams and Assignments

- Week 1: Binary Representation **HW0**
- Week 2: Integer Operations
- Week 3: Floating-Point Operations **Data Lab 1**
- Week 4: Assembly (Arithmetic Instruction)
- Week 5: Assembly (Debugging with GDB) **Data Lab 2**
- Week 6: Assembly (Function Calls)
- Week 7: **Bomb Lab** (Oct. 1), **Exam I** (Oct. 4), Security Vulnerabilities
- Week 8: Memory Organization
- Week 9: Caching **Attack Lab**
- Week 10: Virtual Memory
- Week 11: Dynamic Memory Allocation and Linking (Next Discussion)
- Week 12: **Cache Lab** (**Nov. 5**), Processor Organization, **Exam II** (**Nov. 8**)
- **Week 13: Processor Organization**
- Week 14: Code Optimization and **Thanksgiving**
- Week 15: Cache Coherency **Allocation Lab** and Review
- Week 16: Study Days and **Final** (Dec. 6)

The Allocation Lab

Suggested Roadmap

- Do you have a working implementation? (Start with book implementation.)
- Can you get 40/40 points for throughput? (Try explicit free lists.)
- Can you get at least 40/50 points for utilization?

To get good utilization, look at the traces!

Example

- Allocate 16, 112, 16, 112, 16, … (in this order, contiguously)
- Free the "112" blocks
- Allocate as many "128" blocks

You cannot merge the freed blocks: will end up using 2x space for the heap!

- Any better strategy for placing blocks 16, 112, 16, 112, 16 within free blocks?

Assigned Points

Breakdown

- 25 points for **correctness** (partial credit for each correct trace execution)
- 35 points for **performance**
  - **memory utilization** = peak memory usage / heap size (at most 1)
  - **throughput** = operations / second
  - **performance index** (*w* = 0.6)

$$P = wU + (1 - w) \min\left(1, \frac{T}{T_{libc}}\right)$$

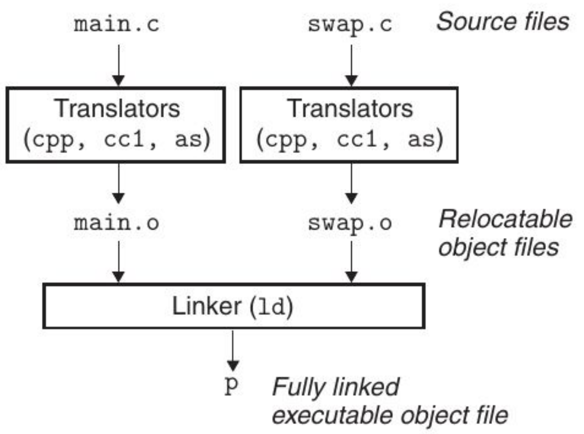
Linking

(a) main.c

```c
/* main.c */
void swap();

int buf[2] = {1, 2};

int main()
{
    swap();
    return 0;
}
```

(b) swap.c

```c
/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```




```
gcc -c main.c swap.c
gcc -o prog main.o swap.o
./prog
```

Storage Class Specifiers

- `extern` ⇒ to declare a global variable/function defined in another unit
- `static` ⇒ to define a global variable/function with internal linkage

Function prototypes are `extern` by default; local variables can be `static` (bad)

Global Variables (Avoid If Possible)

With `extern` specifier

- Cannot initialize the variable (another unit will)
- Expected during linking as a global variable in another unit

With Initialization (Strong Symbol)

- Initialized to the given value
- Exported during linking
  - Linking error if another unit initializes a variable with the same name
  - No error if the other unit defines a weak symbol (no initialization)

Without Initialization (Weak Symbol)

- Initialized to zero if no strong symbol is present
- Exported during linking in “common mode”
  - Shared if another unit defines a variable with the same name

No checks on global variable types: data types may not match (bad!)

- Also no checks on function prototypes of external functions...
- Checks on types/prototypes if linking optimization `-ftlo` is enabled
- Common strategy: each unit includes its own prototypes/externs

External Symbols: When types don't match...

```c
/* main.c */
#include <stdio.h>
int z = 0x11223344;
void swap(int *x, int *y);

int main() {
    int x = 0x11223344;
    int y = 0x55667788;
    printf("z = %x\n", z);
    swap(&x, &y);
    printf("x = %x\n", x);
    printf("y = %x\n", y);
    printf("z = %x\n", z);
}
```

```c
/* swap.c */
short z;

void swap(short *x, short *y) {
    z = 0;
    short tmp = *x;
    *x = *y;
    *y = tmp;
}
```

```
$ gcc -Wall -Wextra -std=c99 main.c swap.c -o prog
$ ./prog
z = 11223344
x = 11227788
y = 55663344
z = 11220000
```

External Symbols: With -flto

```c
/* main.c */
#include <stdio.h>
int z = 0x11223344;
void swap(int *x, int *y);

int main() {
    int x = 0x11223344;
    int y = 0x55667788;
    printf("z = %x\n", z);
    swap(&x, &y);
    printf("x = %x\n", x);
    printf("y = %x\n", y);
    printf("z = %x\n", z);
}
```

```c
/* swap.c */
short z;

void swap(short *x, short *y) {
    z = 0;
    short tmp = *x;
    *x = *y;
    *y = tmp;
}
```

```
$ gcc -Wall -Wextra -std=c99 -flto main.c swap.c -o prog
main.c:4:6: warning: type of ‘swap’ does not match original declaration
[..]
swap.c:1:7: warning: type of ‘z’ does not match original declaration
[..]
main.c:3:5: note: type ‘int’ should match type ‘short int’
```

External Symbols: Using Headers

```c
/* main.h */
#ifndef MAIN_H
#define MAIN_H

#include <stdio.h>
#include "swap.h"
extern int z;

#endif /* MAIN_H */
```

```c
/* swap.h */
#ifndef SWAP_H
#define SWAP_H


#include "main.h"
void swap(int *x, int *y);

#endif /* SWAP_H */
```

```c
/* main.c */
#include "main.h"

int z = 0x11223344;
int main() {
    int x = 0x11223344;
    int y = 0x55667788;
    printf("z = %x\n", z);
    swap(&x, &y);
    printf("x = %x\n", x);
    printf("y = %x\n", y);
    printf("z = %x\n", z);
}
```

```c
/* swap.c */
#include "swap.h"


void swap(int *x, int *y) {
    z = 0;
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

```
$ gcc -Wall -Wextra -std=c99 \
  main.c swap.c -o prog
$ ./prog
z = 11223344
x = 55667788
y = 11223344
z = 0
```

Strategy: Each unit includes its own prototypes/declarations.

- Non-matching types now result in **compile errors** within a unit
- Header guards are used to avoid double (or recursive) inclusion of headers
- Adding `int z = 42` to swap.c results in a **linking error**

Linking

- **Phase 1: Symbol Resolution**
  - **Global Symbols**: Non-static, global variables and functions
  - **External Global Symbols:** Used but not defined in a unit
  - **Local Symbols:** Static variables and functions (used only in this unit)
    - Local variables are not local symbols! (Not involved in linking)
    - Errors for duplicate definition of local symbols,
      duplicate global symbols with initializations (*strong symbols*)
- **Phase 2: Relocation**

main.o

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi
   e:   e8 00 00 00 00          callq  13 <main+0x13>
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

sum.o

```
0000000000000000 <sum>:
   0:   85 f6                   test   %esi,%esi
...
  26:   c6 05 00 00 00 00 01    movb   $0x1,0x0(%rip)
  2d:   c3                      retq
```

Executable Object

```
00000000004004d6 <sum>:
  4004d6:       85 f6                   test   %esi,%esi
  ...
  4004fc:       c6 05 36 0b 20 00 01    movb   $0x1,0x200b36(%rip) # 601039 <done>
  400503:       c3                      retq

0000000000400504 <main>:
  400504:       48 83 ec 08             sub    $0x8,%rsp
  400508:       be 02 00 00 00          mov    $0x2,%esi
  40050d:       bf 30 10 60 00          mov    $0x601030,%edi
  400512:       e8 bf ff ff ff          callq  4004d6 <sum>
  400517:       48 83 c4 08             add    $0x8,%rsp
  40051b:       c3                      retq
```

Symbol Resolution: Global, External, Local

```c
// prototype
int sum(int* a, int n);
// global data
int array[2] = {5, 6};
char done = 0;

int main()
{
  int val = sum(array, 2);
  return val;
}
```

```c
#include <stdio.h>

int x=1, z=0;
static int y=5;

static int foo(int bar)
{
  x += bar;
  y--; z++;
  return x;
}

int main(int argc, char** argv)
{
  printf("%d\n", foo(3));
  return 0;
}
```

Global	array, done, main
External	sum
Local	
Linker-Ignored	val

Global	x, z, main
External	printf
Local	foo, y
Linker-Ignored	argc, argv, bar

Symbol Resolution: Strong and Weak

```
$ gcc res2a.c res2b.c -o res2
```

```
// res2a.c
#include <stdio.h>

void doit(int *sum);

// better: extern int error
int error;
int val;

int main()
{
  doit(&val);
  printf("%d\n", val);
  return 0;
}
```

Strong	main
Weak	error, val

```
// res2b.c
#include <stdio.h>

int error = 0;
int val;

void doit(int *sum)
{
  int x, y;
  if(2 != scanf("%d %d", &x, &y)){
    error = 1;
    return;
  }
  *sum = x+y;
}
// would generate an error, if added
// int main() { return 0; }
```

Strong	doit, error
Weak	val

```
$ gcc -fno-common res2a.c res2b.c
/tmp/ccwo7BuS.o:(.bss+0x0): multiple definition of `error'
/tmp/ccbFljfF.o:(.bss+0x0): first defined here
/tmp/ccwo7BuS.o:(.bss+0x4): multiple definition of `val'
/tmp/ccbFljfF.o:(.bss+0x4): first defined here
collect2: error: ld returned 1 exit status
```

Object Files

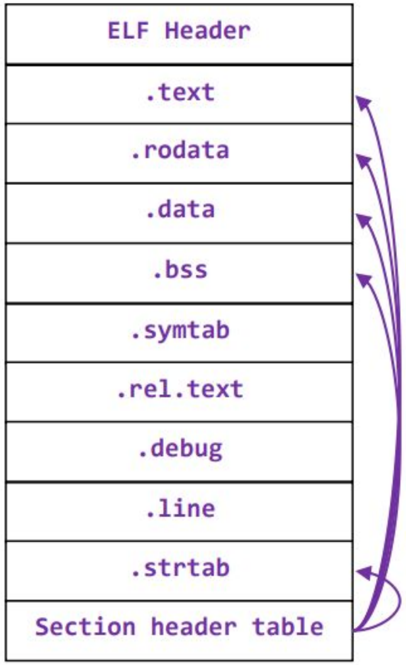
Three kinds of object files:

- **Relocatable:** code/data (e.g., `.o` produced by `gcc -c`)
- **Executable:** binary ready for execution (e.g., ./prog produced by `gcc -o`)
- **Shared:** ready to be used as dynamic library (.so on Linux)

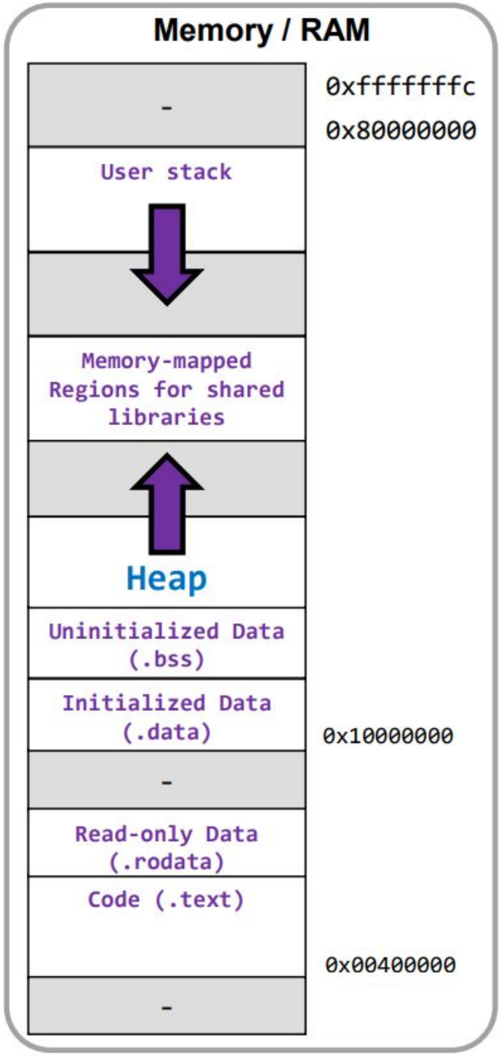
In Linux, executable files have the **ELF format** (Executable & Linked Format)

ELF Format

.text binary code
.rodata constants like strings
.data initialized global/static vars
.bss uninitialized global/static variables (no space in .o)
.symtab symbol table (functions and global variables)
.rel.text relocation info
.debug, **.line**: symbol table for locals and other definitions (included with -g)
.strtab Table of all the strings used by other headers



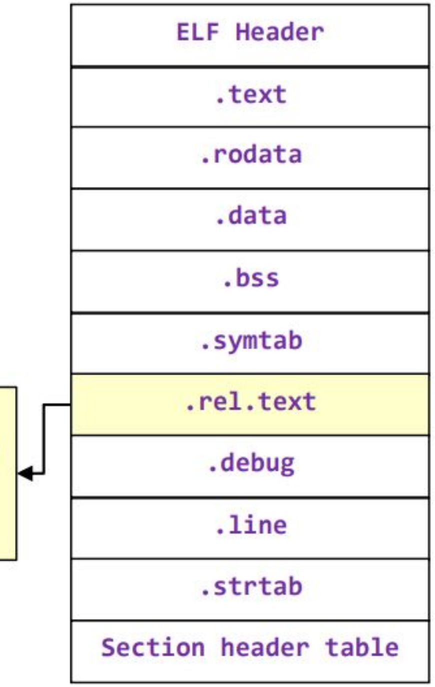
Sample ELF-Header
(Some sections will eventually map directly to memory sections of the executable while others are for bookkeeping purposes to help the linker or loader)



Relocation

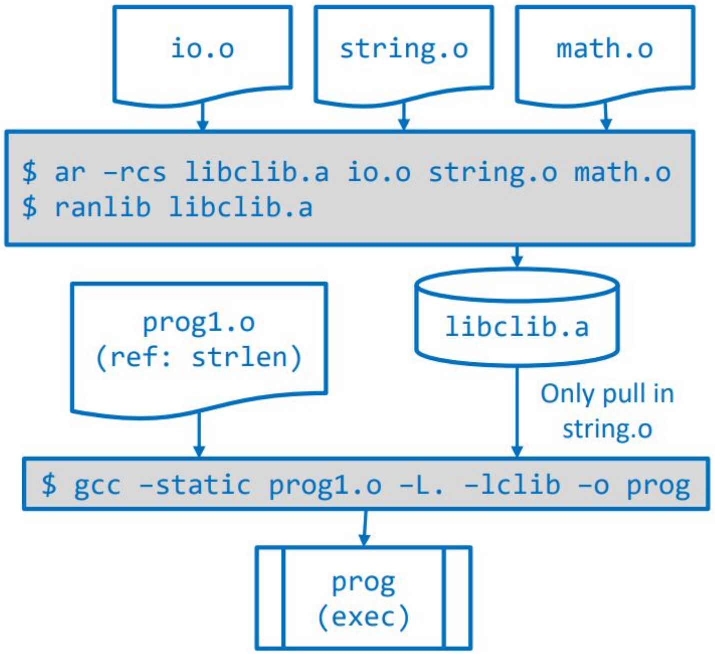
```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi
   e:   e8 00 00 00 00          callq  13 <main+0x13>
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

```
Relocation section '.rela.text' at offset 0x208 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000a  000900000000a R_X86_64_32       0000000000000000 array + 0
00000000000f  000a00000002 R_X86_64_PC32     0000000000000000 sum - 4
```



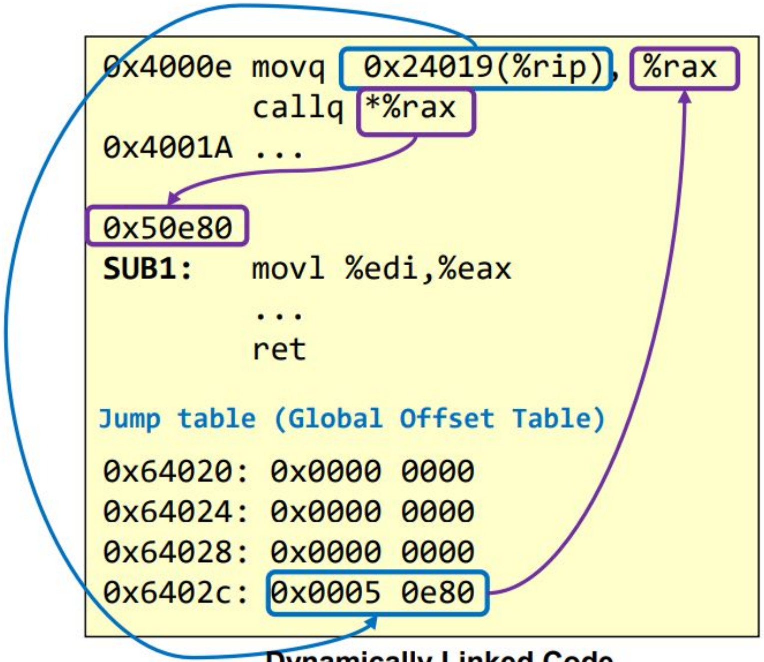
Static Libraries

- Binary code from library functions added to executable at linking time
- Only needed functions are included
- No dependencies at runtime
- Large executable
- Each program loads the same library code into memory
- Need to run the linker again to use a new library version (e.g., with bug fixes)



Shared Libraries

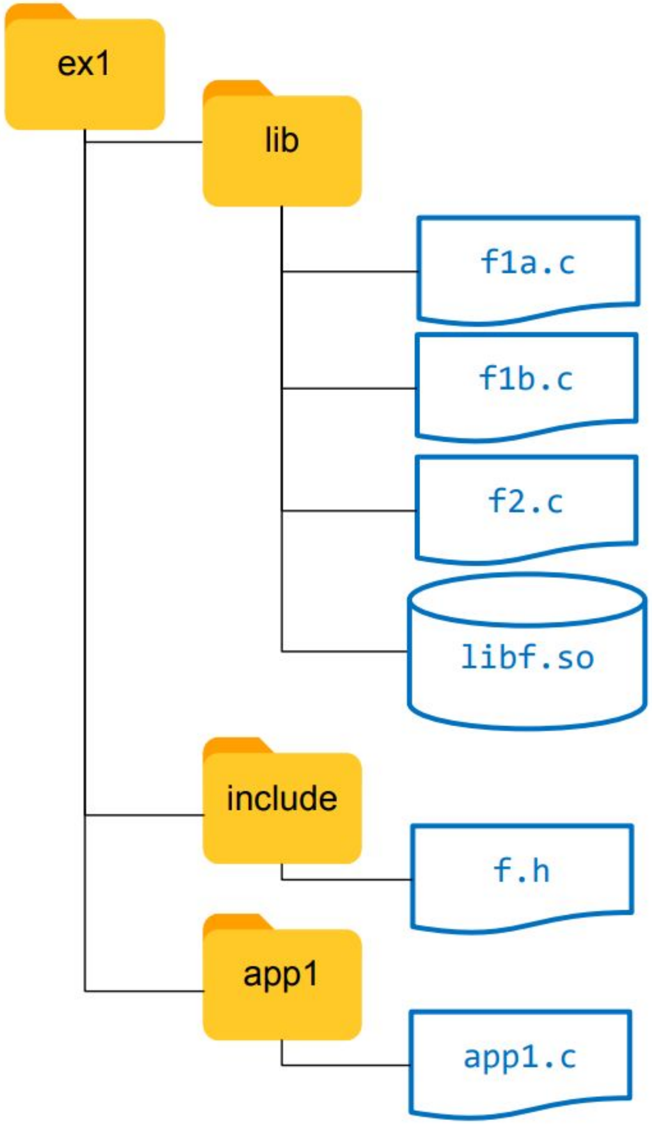
- Use an indirection: lookup address of code/data at runtime
- OS loader fills Global Offset Table with runtime location of code
- Many processes can share the same code (as read-only areas)
- If system libraries are updated, a new version is used
- Compile with `-shared`



Dynamically Linked Code

Example

```
cd lib
rm -f *.o *.a
gcc -c -fpic f1a.c f2.c
gcc -shared f1a.o f2.o -o libf.so
ls
cd ../app1
gcc -I../include -L../lib app1.c -lf
./a.out       # loader can't find libf.so
# set search path for libraries
export LD_LIBRARY_PATH=../lib:$LD_LIBRARY_PATH
./a.out       # should see '11' output
cd ../lib
rm libf.so
gcc -c -fpic f1b.c
gcc -I../include -shared f1b.o f2.o -o libf.so
cd ../app1
./a.out       # should se '1111' output
              # without recompile/relink
cd ..
```



Processor Families

Instruction Set Architecture (ISA)
Instructions supported by a processor (and their byte-level encoding).

- Examples: x86-64, IA32, ARMv8.

Processor Family
Different processors implementing the same ISA.

- Examples: Intel i5 and i7 (x86-64).

The ISA is the shared interface / level of abstraction for:

- Compiler writers (translate C to assembly of an ISA).
- Processor designers (design logic to execute ISA assembly instructions).

Very clever optimizations are adopted by processor designers:

- Pipeline
- Out-of-order execution
- Branch prediction

Recently responsible of security attacks (Meltdown and Spectre).

Main Idea: Parallelism

Take sequential ISA instructions and run them in parallel.

- The result must be the same as sequential execution.

Parallelism at many levels

- At sub-instruction level: pipeline.
- At instruction level: superscalar execution (e.g., two pipelines).
- At thread level: run multiple threads on separate cores.
- At data level: single-instruction multiple-data (SIMD).

Problems

- Data dependencies: the next instruction needs (at some point) the result of the previous one. Cannot run them in parallel!

Clever strategies to deal with data dependencies:

- Out-of-order execution
- Static and dynamic scheduling
- Loop unrolling and renaming

Instruction Sets: RISC and CISC

CISC Processors

- Large number of instructions
- Instructions with long execution time (e.g., memory to memory)
- Complex, variable-size instruction encodings (e.g., 1-15 bytes for x86-64)
- Complex addressing formats, e.g., `movq %rds,2(%rax,%rdx,8)`
- ALU operations applicable to memory and registers: `addq %rcx,(%rax)`
- Stack intensive: use stack for return addresses and arguments (e.g., IA32)

RISC Processors

- Many fewer instructions (less than 100)
- Instructions only for quick, primitive operations
- Fixed-length instruction encoding (typically, 4 bytes)
- Simple addressing formats, e.g., just base and displacement: `2(%rax)`
- ALU operations applicable only to registers: `addq %rcx,%rax`
- Register intensive: use registers for return addresses and arguments.

Today: x86-64 CISC instructions translated by CPU to RISC-like instructions.

Example: Translating to RISC-like assembly

```
// CISC instruction
movq 0x40(%rdi, %rsi, 4), %rax

// RISC equivalent
mov   %rsi, %rbx   // use %rbx as a temp
shl      2, %rbx   // %rsi * 4
add   %rdi, %rbx   // %rdi + (%rsi*4)
add  $0x40, %rbx   // 0x40 + %rdi + (%rsi*4)
mov (%rbx), %rax   // %rax = *%rbx
```

General Principles

- Replace complex addressing with sequence of arithmetic operations
- Replace memory-to-register ALU operations with register-to-register operations and load/store.

RISC: Classroom Instructions

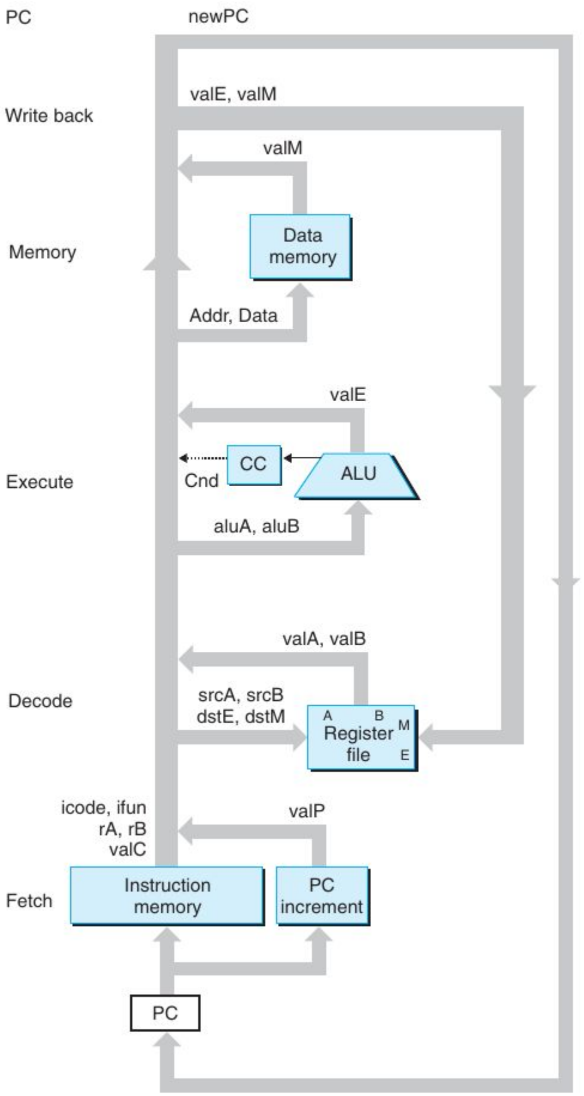
- Load from memory into register:
  - `ld 0x40(%rdi), %rax`
- Store register into memory:
  - `st %rax, 0x40(%rdi)`
- Arithmetic and logic instructions on registers:
  - `add %rdi, %rax`
  - `sub %rdi, %rax`
  - `xor %rdi, %rax`
  - `...`
- Moves between registers
  - `mov %rdi, %rax`
- Jumps
  - `je 0x123`
  - `jg 0x123`

Example: Translation

```
// example #1
mov (%rdi), %rax
mov 0x40(%rdi), %rax
mov 0x40(%rdi,%rsi), %rax

// example #2
mov %rax, (%rdi)
mov %rax, 0x40(%rdi)
mov %rax, 0x40(%rdi,%rsi)

// example #3
add %rax, (%rsp)
```

```
ld 0x0(%rdi), %rax
ld 0x40(%rdi), %rax
mov %rsi, %rbx
add %rdi, %rbx
ld  0x40(%rbx), %rax

st %rax, 0x0(%rdi)
st %rax, 0x40(%rdi)
mov %rsi, %rbx
add %rdi, %rbx
st  %rax, 0x40(%rbx)

ld   0(%rsp), %rbx
add  %rax, %rbx
st   %rbx, 0(%rsp)
```

Sequential Processor



On each clock cycle, perform all the steps to run an instruction (so, clock cycle will be large!).

Fetch. Read instruction from memory and extract `icode`, registers `rA`/`rB`, constant `valC`.

Decode. Read up to 2 operands from register file, obtaining `valA` and `valB` (for ALU operations).

Execute. ALU operation on registers, effective address computation (for `ld` and `st`). Produces an output value and a condition code.

Memory. Read data from memory to `valM` (for `ld`), or write data to memory (for `st`). Uses the address computed during Execute.

Write Back. Save Ex/Mem output to registers.