Lecture 3: Runtime Environment

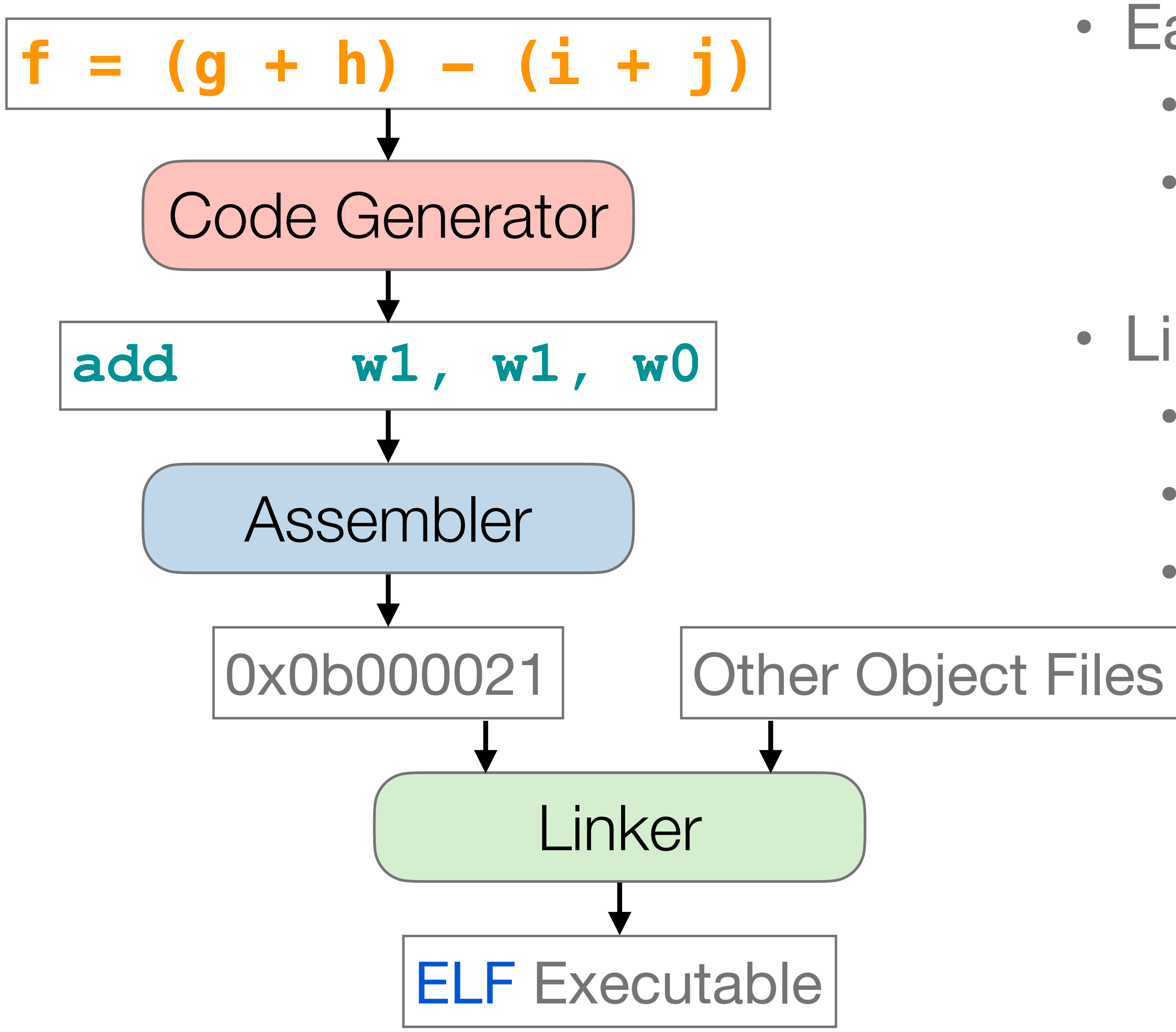
Spring 2024
Jason Tang

Topics

- Program startup
- Stack pointer
- Instruction density

Compilation Process

- A "compiler" is a collection of tools that transform program code into executable binaries



- Each object file contains:
  - File header
  - Index of sections (file offsets)
- Linker Tool:
  - Merges together all object files
  - Resolves all internal symbols
  - Writes resulting executable

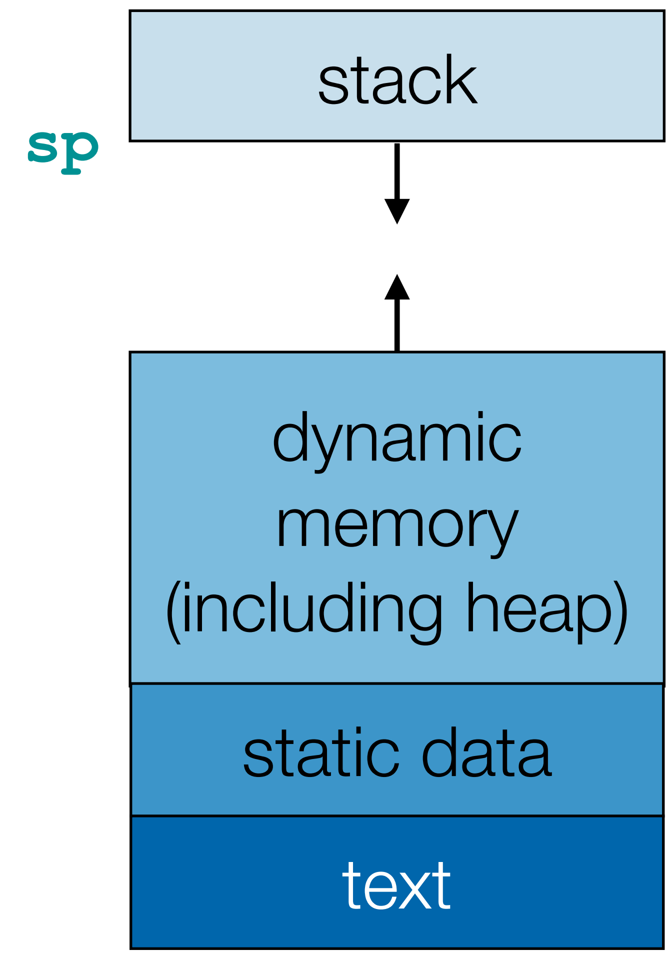
Program Segments

- ELF executable contains several segments:
  - code segment: machine code, set as read-only and executable
  - data segment: global variables
    - rodata segment: read-only data
    - BSS (block started by symbol): initialized to zero at startup

Program Loader

- Part of the operating system that loads the executable file from storage and allocates space in memory for code and data segments

Typical Memory Layout



  - Lays out memory regions
  - Explicitly zeroizes BSS region
  - Initializes registers, especially the stack pointer
    - ARMv8's stack pointer is `sp`
  - Jumps to entry point

ELF File Header

```
$ readelf -h ./test_executable
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x400540
  Start of program headers:          64 (bytes into file)
  Start of section headers:          9064 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         35
  Section header string table index: 32
```

ELF Program Header

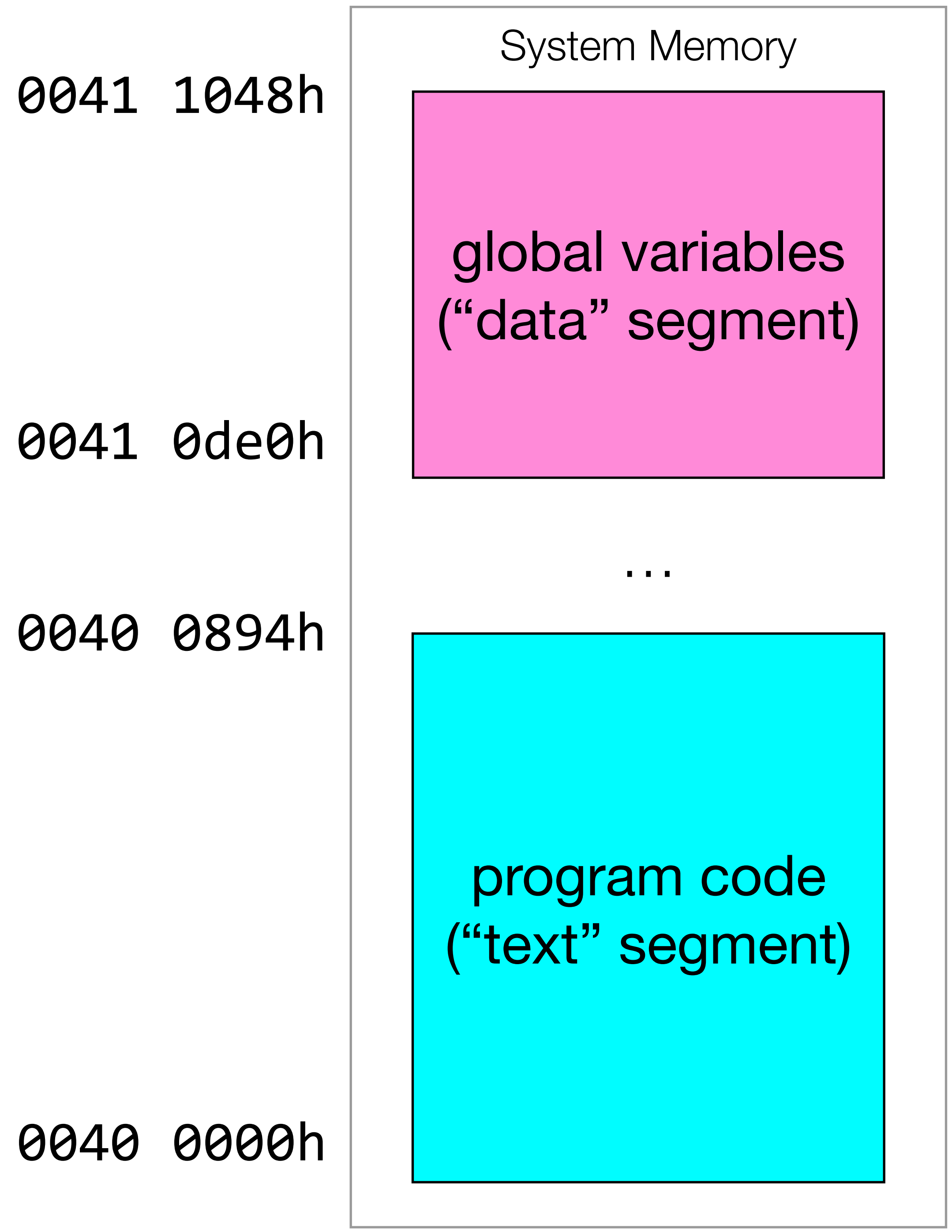
```
$ readelf -l ./test_executable

Elf file type is EXEC (Executable file)
Entry point 0x400540
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R E    8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001b 0x000000000000001b  R      1
      [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000894 0x0000000000000894  R E    10000
  LOAD           0x0000000000000de0 0x0000000000410de0 0x0000000000410de0
                 0x0000000000000258 0x0000000000000268  RW     10000
  DYNAMIC        0x0000000000000df8 0x0000000000410df8 0x0000000000410df8
                 0x00000000000001e0 0x00000000000001e0  RW     8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000de0 0x0000000000410de0 0x0000000000410de0
                 0x0000000000000220 0x0000000000000220  R      1
```

Example Memory Layout

```
Type           VirtAddr           PhysAddr
               MemSiz              Flags  Align
PHDR           0x0000000000400040 0x0000000000400040
               0x00000000000001c0  R E    8
INTERP         0x0000000000400200 0x0000000000400200
               0x000000000000001b  R      1
LOAD           0x0000000000400000 0x0000000000400000
               0x0000000000000894  R E    10000
LOAD           0x0000000000410de0 0x0000000000410de0
               0x0000000000000268  RW     10000
DYNAMIC        0x0000000000410df8 0x0000000000410df8
               0x00000000000001e0  RW     8
```



ARMv8-A C Runtime Environment (CRT)

- Allocates space for text and data segments, then copies from executable file into those spaces
- Clears BSS
- Copies runtime arguments into registers (`x0`, `x1`, etc.)
- Allocates space for stack and sets `sp` to the topmost address
- Sets link register (`x30`) to return address
- Unconditional branch to entrypoint (e.g., `main()`)

https://git.pengutronix.de/cgit/barebox/tree/arch/arm/cpu/setupc.S

AArch64 Stack Pointer

- Stack always grows downwards
- `sp` holds address of lowest valid value on the stack
- `sp` must be aligned on a 16 byte boundary
  - If the size of the item to store is not evenly divisible by 16 bytes, then `sp` must be further decreased to stay on a 16 byte boundary
- By convention, `x29` is the frame pointer
  - Holds previous value of sp when entering a function

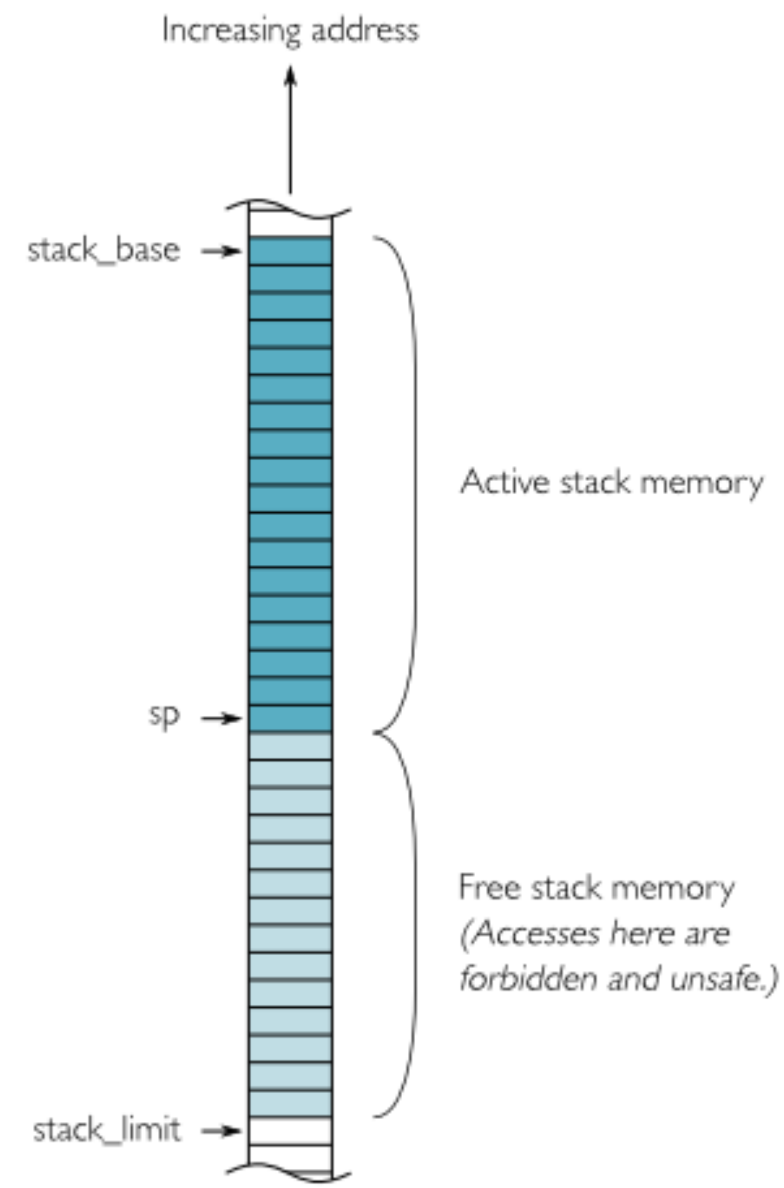


Diagram from https://community.arm.com/processors/b/blog/posts/using-the-stack-in-aarch32-and-aarch64

Nested Functions

- When a function is about to be called, the caller must preserve registers it will use after function has completed
  - Typically, need to push the link register (**`x30`**) and frame pointer (**`x29`**) on to stack
- Called function (the callee) must restore registers prior to leaving function
  - Restore caller's link register and frame pointer

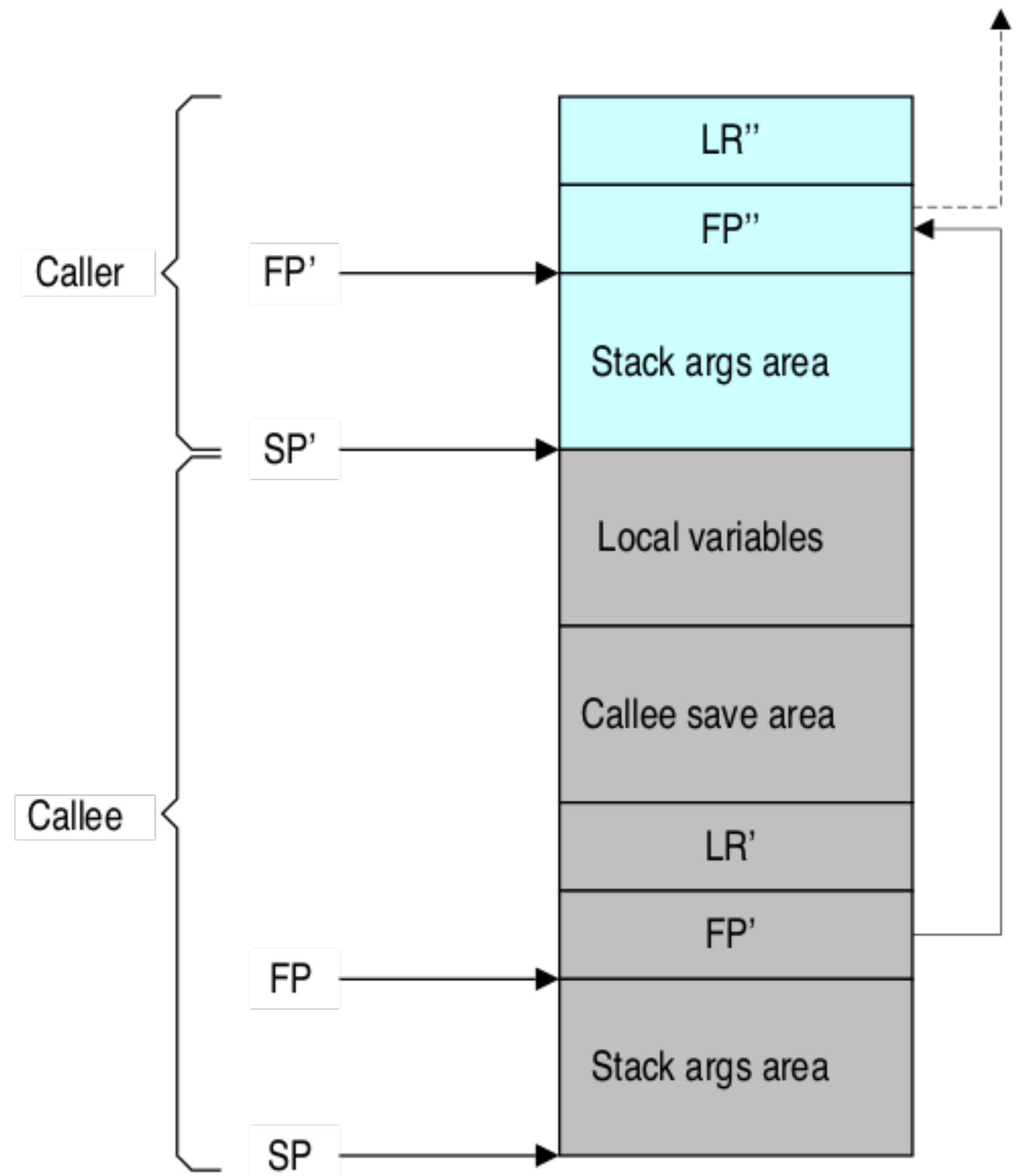


Diagram based upon *Procedure Call Standard for the ARM 64-bit Architecture*, §B.3

Nested Functions

```c
#include <stdio.h>
static int __attribute__((noinline))
 func(int x) {
    return printf("%d\n", x);
}

int main(int argc, char *argv[]) {
    func(argc);
    return 0;
}
```

```
func:
  stp    x29, x30, [sp,#-32]!
  mov    x29, sp
  str    w0, [x29,#28]
  adrp   x0, 400000 <_init-0x3f0>
  add    x0, x0, #0x6a0
  ldr    w1, [x29,#28]
  bl     400460 <printf@plt>
  ldp    x29, x30, [sp],#32
  ret
main:
  stp    x29, x30, [sp,#-32]!
  mov    x29, sp
  str    w0, [x29,#28]
  str    x1, [x29,#16]
  ldr    w0, [x29,#28]
  bl     4005c0 <func>
```

ARMv8-A Procedure Call Standard

Register	Special	Role
X31	SP	Stack Pointer
X30	LR	Link Register
X29	FP	Frame Pointer
X19 - X28		Callee-saved registers
X18		“Platform Register”
X17	IP1	“Intra-Procedure Call” Register
X16	IP0	“Intra-Procedure Call” Register
X9 - X15		Temporary Registers
X8		Indirect Result Location Register
X0 - X7		Parameter / Result Registers

Table based upon *Procedure Call Standard for the ARM 64-bit Architecture*, §5.1.1

Instruction Constraints

- Every instruction performs one operation
- On ARMv8-A, [most] instructions fit in 32 bits
  - Top 10 or 11 bits specify which opcode to perform
  - Remaining bits give operands to instructions
  - Because there are 31 usable general purpose registers, need 5 bits to encode a register number
- Insufficient space in a single instruction to specify a constant consisting of more than 16-ish bits

Immediate Operands

- Use of constants is common in programs (e.g., incrementing an index in an array, number of loop iterations, etc.)
  - 52% of arithmetic operands in the *gcc* program involve constants*
- Two approaches:
  - Store constant in memory, then load it into a register (slower)
  - Use two instructions, one to write lower bits and another to write upper bits (faster)
    - Because most operations involve small constant values, the second instruction is often omitted by the compiler

* According to Textbook

ARMv8-A Immediate Values

- In ARMv8-A, **`mov`** is used to write a 16-bit immediate value and zeroes out remaining bits in target register

- **`movk`** instruction shifts the 16-bit immediate value when writing, keeping other bits unchanged

```
int x = 0x12345678;
```

→

```
mov     w0, #0x5678
movk    w0, #0x1234, lsl #16
```

- Can use multiple **`movk`** instructions to write each 16 bits of a 64-bit register

ARMv8-A Branching

- Unconditional branching (`b`, `bl`) sets PC to value relative to current address

B-Type	opcode	immediate
	6 bits	26 bits

  - Because all instructions must be word-aligned, the immediate value is multiplied by 4, and then added to the current program counter*

- Conditional branching (`cbz`, `cbnz`, and `b.`*`cond`*) also sets PC to a relative value (multiplied by 4), but its immediate field is smaller

CB-Type	opcode	immediate	R_t
	8 bits	19 bits	5 bits

  - Most conditionals involve nearby jumps, while function calls can be far away

  - If need to jump very far away, then store target address in a register and use `blr`

* Technically not true. See https://stackoverflow.com/questions/24091566/why-does-the-arm-pc-register-point-to-the-instruction-after-the-next-one-to-be-e

Classifying Instruction Set Architectures

- Accumulator Architecture: common in early stored program computers when hardware was expensive
  - Only one register for all arithmetic and logical operations
  - All operations use accumulator as a source operand and as destination; all other operands stored in memory
- Extended Accumulator Architecture: dedicated registers for specific operations

Traditional x86 Register Uses (Simplified)		
AX	accumulator	used for arithmetic
BX	base	base pointer for memory access
CX	counter	loop counter
DX	data register	used for arithmetic

Classifying Instruction Set Architectures

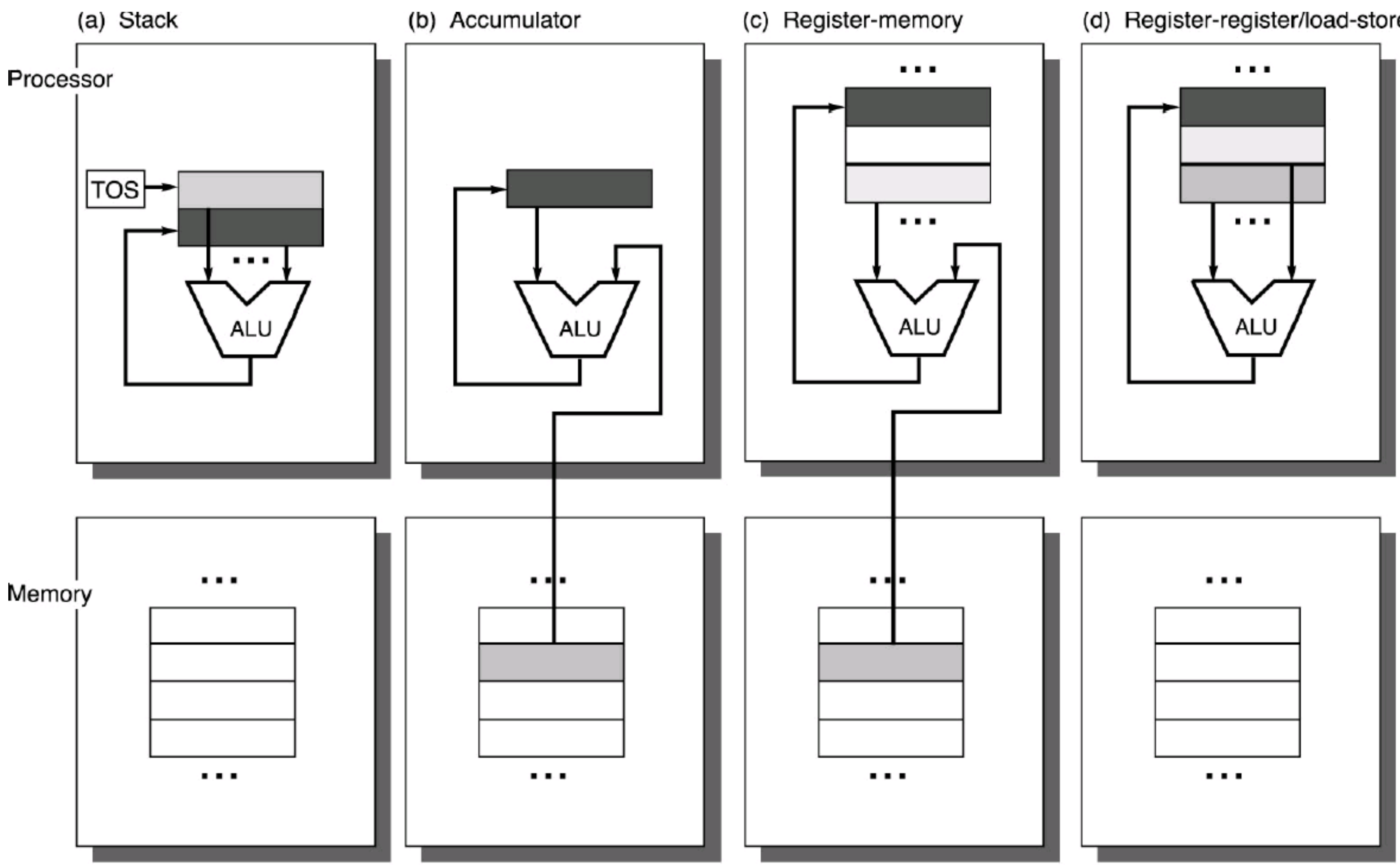
- General-Purpose Register Architecture: (nearly any) register can be used for any purpose
  - Register-memory: allows one operand to be a memory address
  - Register-register (load-store): all operands must be registers

Machine	Number of GPRs	Architecture Style	Year
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
PowerPC	32	Load-store	1992

Instruction Density

- Variable-length architectures (such as x86) is good when memory is scarce
    - Minimizes code size, leading to higher instruction density per byte
- Stack machines abandoned registers altogether
    - Operands pushed on to a memory stack, then popped to perform operation, then results pushed back to stack
    - Extremely small instructions, though more instructions needed per operation
    - Simplifies compiler construction, when compilers were non-optimizing

Instruction Set Architecture Designs



CISC vs. RISC

- Complex Instruction Set Computer (CISC):
  - In 1960s, software was usually written in assembly, not in a high-level language
  - Instructions were added to mimic high-level constructs
  - Higher code density, but made hardware more complex
- Reduced Instruction Set Computer (RISC):
  - Reduced number of instructions that hardware must implement
  - Relied on compiler to effectively use hardware

CISC vs. RISC Debate

CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple operations per instruction	Single operation per instruction
High code density, variable length instructions	Low code density, fixed width instructions
Transistors used to store instructions, not registers	Transistors used on memory registers

- Modern architectures are based on both principles
    - x86 was originally pure CISC, x86-64 is RISC-like internally
    - ARMv8-A is mostly RISC, but its math extensions are CISC-like

https://www.eetimes.com/author.asp?doc_id=1327016

Principles of Hardware Design

- Make common case fast
  - Example: immediate field widths are sized for most common cases
- Smaller is faster
  - A CPU with more registers is harder to build, physically
- Good design demands good compromises
  - RISC simplifies instruction decoding at the expense of more instructions to decode