# Lecture 2: Instruction Set Architecture (ISA) part.1

Jangwoo Kim (Seoul National University)

jangwoo@snu.ac.kr

Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch @ Carnegie Mellon University, University of Michigan, Purdue University, University of Pennsylvania, University of Wisconsin, POSTECH, and SNU

## Office Hour

- Office Hour
  - Schedule your meeting with TAs and me
  - For any matters, we will welcome you

- URL
  - Use **eTL** actively!
    - Lecture slides will be uploaded
    - Use the Q&A board as much as possible
    - If you have a question, your friends must have the one too.

## Announcement

- **Make-up Lecture (this week only!)**
  - 3/8 (Thursday 9:30AM) Lecture #2
    → 3/6 (Tuesday 5PM) Lab #1
  - 3/6 (Tuesday 5PM) Lab #1
    → 3/8 (Thursday 9:30AM) Lecture #2

Sorry for this sudden notice. :<

## [Important] Auditing Policy

- **Too many "audit" requests!!**
  - Students from the other session
  - Students interested in doing lab only
  - Students wanting to study in advance
  - Graduate students in other fields

**Basic Policy:**

**(1) If not registered nor audited, DO NOT attend lecture.**

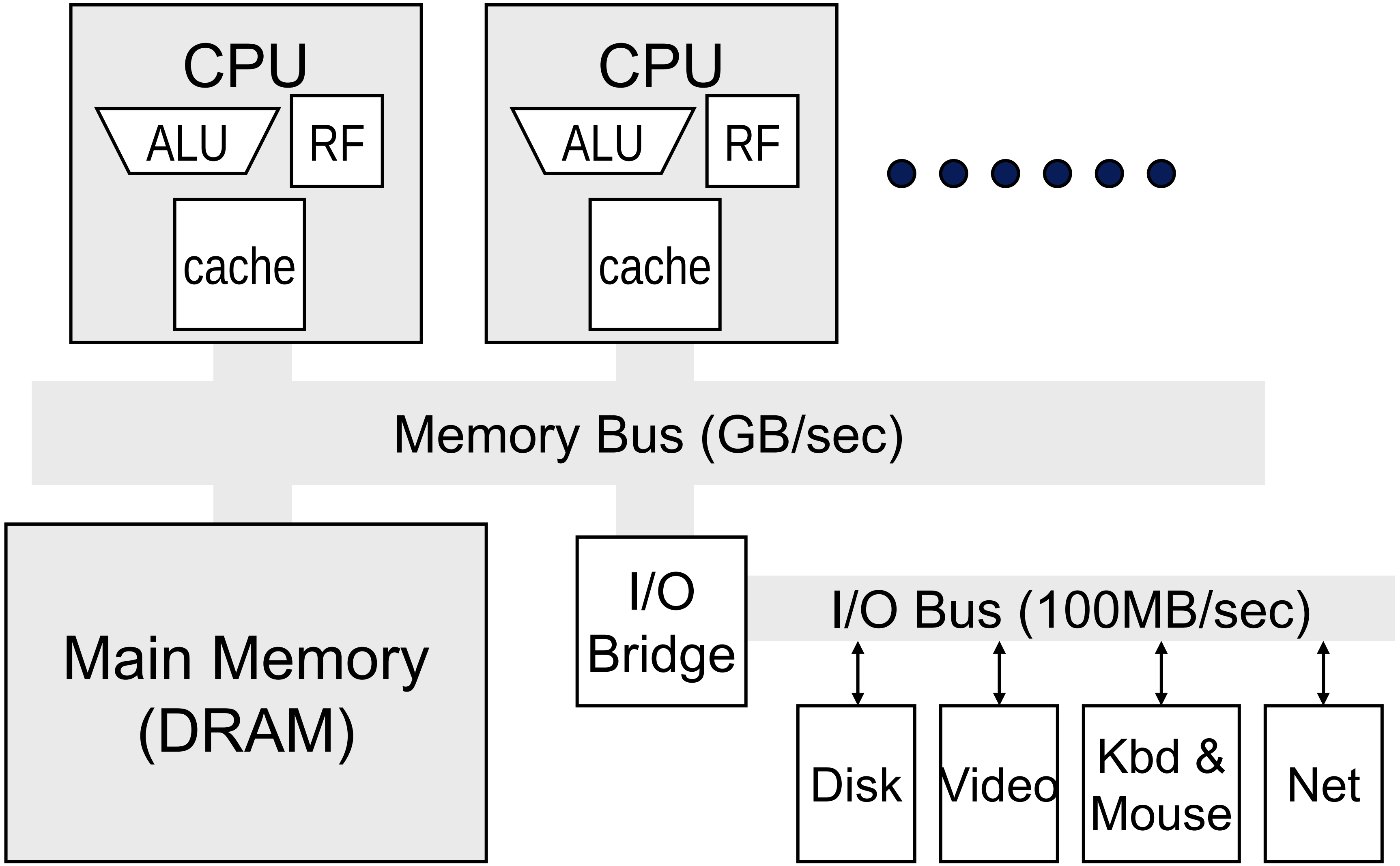
**(2) Audit will be allowed only for graduate students (or some students who I specifically allow)**

**(3) Audit students should not take exam nor do lab**

## Course Plan

- 25+ lectures
  - Lec .1 ~ 12 : **Microprocessors** (e.g., ISA, CPU structure, Pipelining)
  - Lec. 13 ~ 20: **Memory Hierarchy** (e.g., Virtual memory, Cache)
  - Lec. 20 ~ end : **Multi-processors** (e.g., Multi-core, cache coherence, GPU)
- Lecture + homework
  - How modern computer systems work?
  - How to design and evaluate a design?
- Programming:
  - How to simulate a chip to finalize a design?
- Lab:
  - How to implement the design on a real chip?

## How to specify what a computer does?



## How to specify what a computer does?

- **Architecture** Level
  - Car: driving manual & operation manual
    // you don't have to be a car mechanic to drive a car.
  - Computer : ?
    // you don't have to be a circuit designer to use a computer.
- **Microarchitecture** (implementation) Level
  - A particular car design has a certain configuration of electrical and mechanical components (e.g., v8 engine vs. v4 engine)
  - A particular computer design has a certain configuration of datapath and control logic units (e.g., 4M cache vs. 1M cache)
- Circuit (=physical implementation) level
  - Car: metal sheets, cranks, shafts, gears, ….
  - Computer: gates, wires, semiconductor, transistors, …

# Architecture*

- "The term *architecture* is used here to describe the attributes of a system **as seen by the programmer**, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation."

--- footnote on page 1, Architecture of the IBM System/360, Amdahl, Blaauw and Brooks, 1964.

## Stored Program (von Neumann) Architecture

- Stored-program architecture
  - Instructions in a linear memory array
    - Instructions (as well as data) are in memory
  - Instructions can be modified just like data
    - Both forms consist of 0s and 1s

- Sequential instruction processing
  - <u>Program counter</u> identifies the current instruction
  - Instruction is fetched from memory and executed
  - <u>Program counter</u> is advanced (according to instruction)
  - Repeat

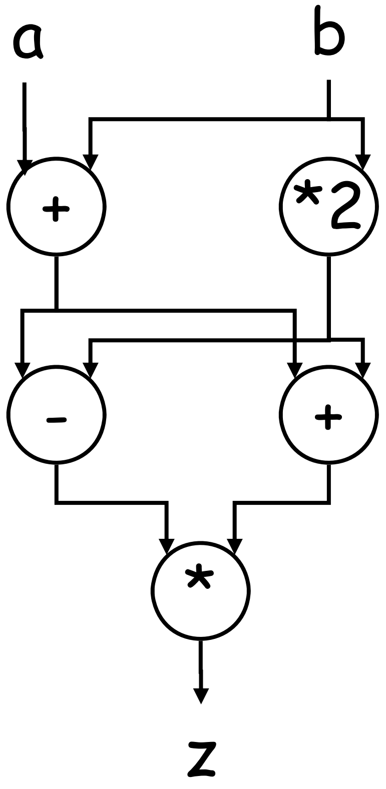
Burks, Goldstein, von Neumann, Preliminary discussion of the logical design of an electronic computing instrument,1946.

# von Neumann vs Dataflow

- von Neumann: Consider a von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations & computations?

"von Neumann bottleneck"

```
v <= a + b;
w <= b * 2;
x <= v - w
y <= v + w
z <= x * y
```



- Dataflow: Instruction ordering specified by dataflow dependence (no program counter!!)
  - Each instruction specifies who should receive results
  - An instruction can execute "whenever" all operands are received

Which one is better?

# Instruction Set Architecture (ISA)

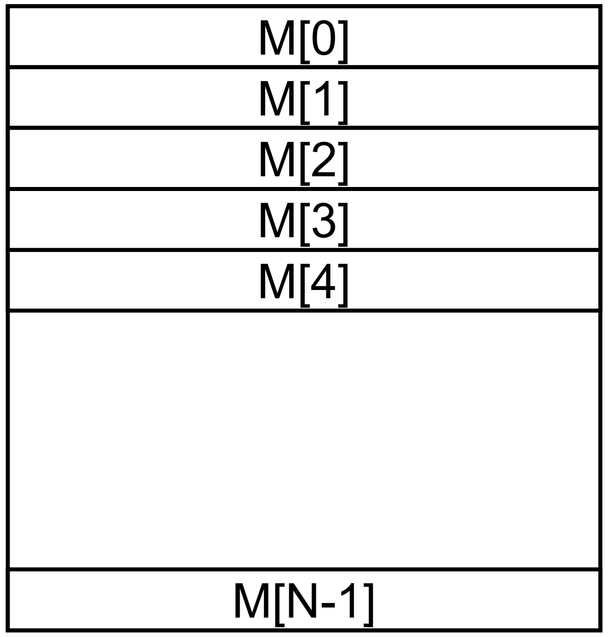
## “User’s manual for the computer”

## What are specified/decided in an ISA?

- Data format and size
  - character, binary, decimal, floating point, negatives
- "Programmer Visible State" **(a.k.a. architectural state)**
  - memory, registers, program counter (PC), etc.
- Instructions: how to **"change"** the programmer visible state?
  - What to perform and what to perform next
  - Where the operands are
- Instruction-to-binary (or vice versa) encoding
- How to interface with the outside world?
- Protection and privileged operations
- Software conventions

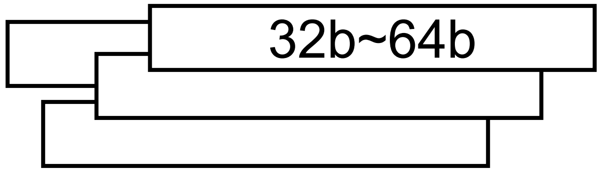
Very often you compromise immediate optimality for future scalability and compatibility

## Programmer Visible State



Memory

Array of storage locations indexed by an address



Registers

- Given special names in the ISA (as opposed to addresses)
- General vs. special purpose

Program Counter (32b~64b)

Memory address of the current instruction

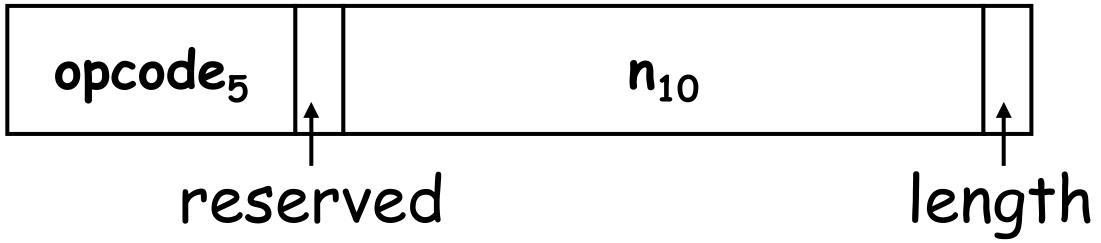
Instructions (and programs) specify how to transform the values of programmer visible state

# General Instruction Classes

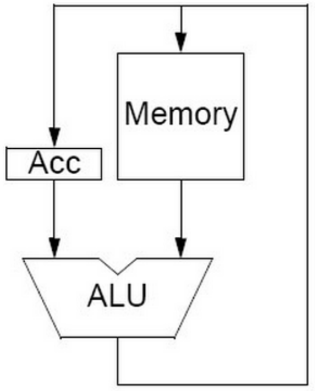
- Arithmetic and logical operations (e.g., add, sub, and, or)
  1) Fetch operands from specified locations
  2) Compute a result as a function of the operands
  3) Store result to a specified location
  4) Update PC to the next sequential instruction
- Data movement operations (e.g., move, load, store)
  1) Fetch operands from specified locations
  2) Store operand values to specified locations
  3) Update PC to the next sequential instruction
- Control flow operations (e.g., branch, jump)
  1) Fetch operands from specified locations
  2) Compute a branch condition and a target address
  3) If "branch condition is true" then PC $\leftarrow$ target address
     else PC $\leftarrow$ next seq. instruction

Generally defined to be atomic.

## An Early ISA: EDSAC (~1950)



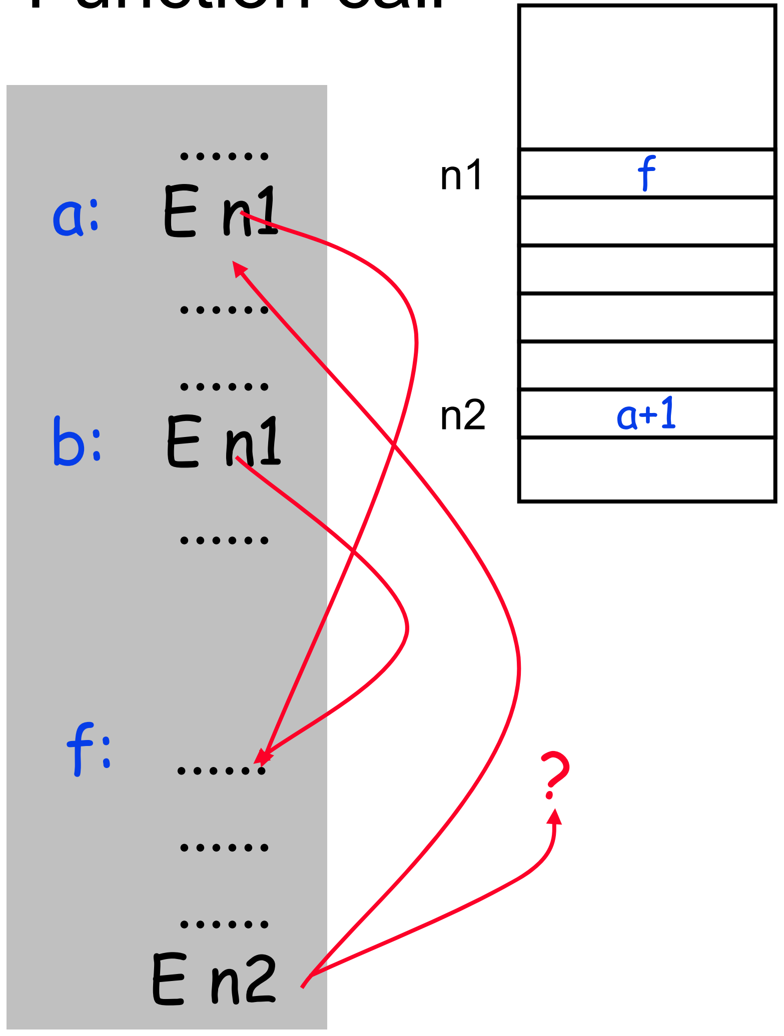
- Single accumulator architecture, i.e. ACC ⇐ ACC ⊕ M[n]
  - **A n**: add M[n] into ACC
  - **T n**: transfer the contents of ACC to M[n]
  - **E n**: If ACC>-1, branch to M[n] or proceed serially
  - **I n**: Read the next character from paper tape, and store it as the least significant 5 bits of M[n]
  - **Z**: Stop the machine and ring the warning bell



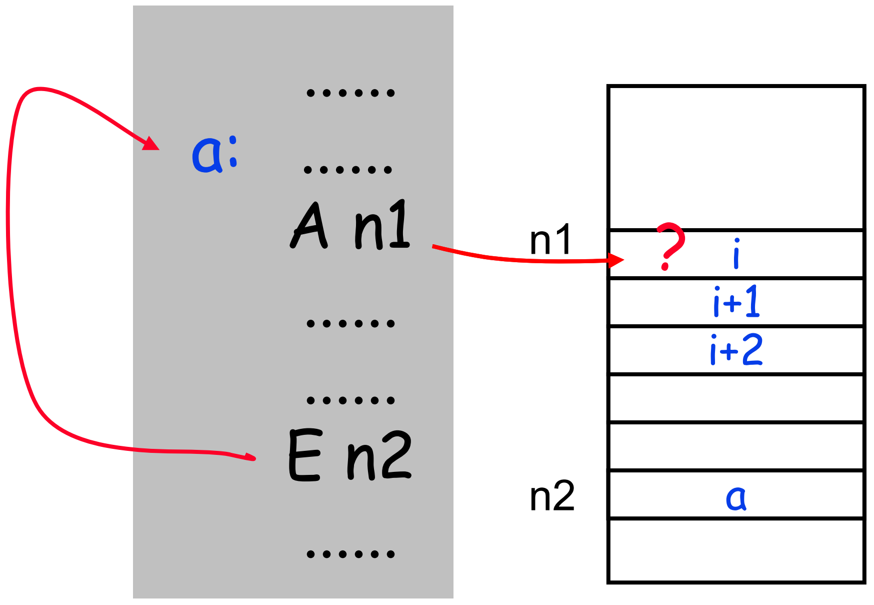
- Only "absolute" addressing in data and control transfer

## Would you make the same mistake?

- Function call



- Array access in a loop



How to fix this?

# Evolution of "Register" Architecture

- Accumulator
  - A legacy from the "adding" machine days

    "AC" button on your calc?
- Accumulator + address registers
  - Need register indirection
  - Initially address registers were special-purpose,

    i.e., can only be loaded with an address for indirection
  - Eventually arithmetic on addresses became supported
- "General purpose registers (GPR)"
  - **All registers good for all purposes**
  - Grew from a few registers to 32 (common for RISC) to 128 (e.g., Intel Itanium)

What are driving the changes?

## Operand Sources?

- Number of Operands

| | | |
|---|---|---|
| **Monadic** | OP in2 | (e.g. EDSAC) |
| **Binatic** | OP inout, in2 | (e.g. IBM 360) |
| - 3 operands in a smaller encoding to save memory | | |
| **Triadic** | OP out, in1, in2 | (e.g. MIPS) |

- Can ALU operands be in memory?

| | |
|---|---|
| **Yes!** | e.g. x86/VAX/"CISC" |
| **No!** | e.g. MIPS/"RISC" → "**load-store architecture**" |

- How many different variations

| | |
|---|---|
| **a very few** | e.g. MIPS / RISC |
| **a lot** | e.g. x86 |
| **everything goes** | e.g. VAX |

# Memory Addressing Modes

- Absolute LW rt, 10000 // LW=load word
  use immediate value as address
- Register Indirect: LW rt, ($r_{base}$)
  use GPR[$r_{base}$] as address
- Displaced or based: LW rt, offset($r_{base}$)
  use offset+GPR[$r_{base}$] as address
- Indexed: LW rt, ($r_{base}$, $r_{index}$)
  use GPR[$r_{base}$]+GPR[$r_{index}$] as address
- Memory Indirect LW rt (($r_{base}$))
  use value at M[ GPR[ $r_{base}$ ] ] as address
- Auto inc/decrement LW rt, ($r_{base}$) ; // $r_{base}$ ++/--
  use GRP[$r_{base}$] as address, but inc. or dec. GPR[$r_{base}$] each time
- Anything else can you think off ......

# VAX-11: ISA in mid-life crisis

- First commercial 32-bit machine

  considered an important milestone

- Ultimate in "orthogonality" and "completeness"

  All of the above addressing modes x { 7 integer and 2 floating point formats} x {more than 300 opcodes}

- Opcode in excess
  - 2-operand and 3-operand versions of ALU ops
  - INS(/REM)QUE (for circular doubly-linked list)
  - "polyf": 4th-degree polynomial solve

- Encoding

  addl3 r1,737(r2),(r3)[r4]    7-byte, sequential decode

## MIPS RISC

- Simple operations
  - 2-input, 1-output arithmetic and logical operations
  - Only few alternatives exist to do the same thing
- Simple data movements
  - ALU ops are register-to-register (need a large register file)
  - Memory can be accessed by only load and store instructions
    → **"Load-store architecture"**
- Simple branches
  - Limited varieties of branch conditions and targets
- Simple instruction encoding
  - All instructions encoded **in the same number of bits**
  - Only a few formats

Such ISA intended for compiler advances
rather than assembly programmers

## Evolution of ISA

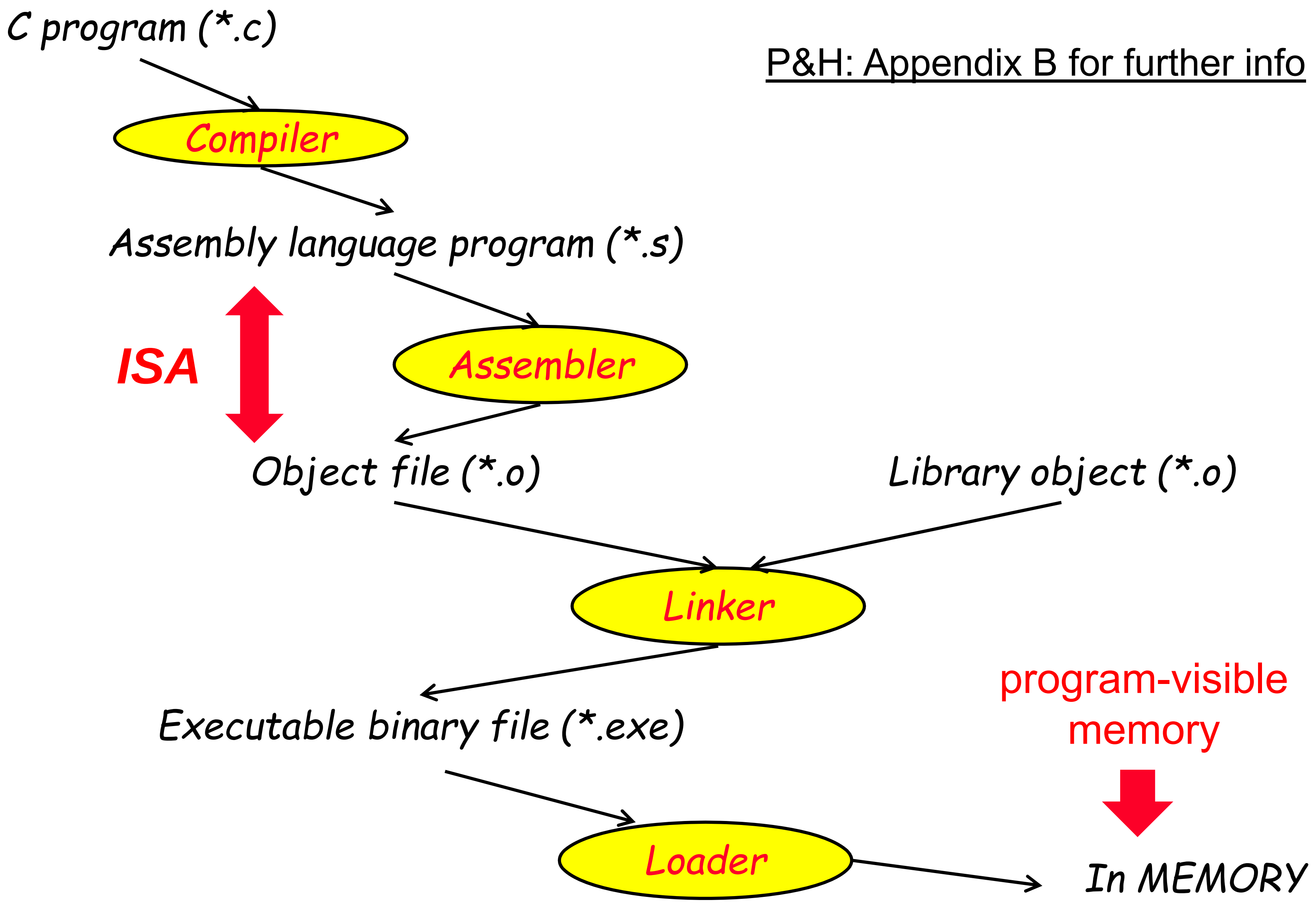
- Why were the earlier ISAs so simple? e.g. EDSAC
  - Technology limitation
  - Inexperience, lack of precedence
- Why did it get so complicated later? e.g. VAX11
  - Assembly programming
  - Lack of memory size and performance
  - Microprogrammed implementation
- Why did it become simple again? e.g. RISC
  - Memory size and speed (cache!)
  - **Compilers**
- Why x86 is still "king of the hill"?
  - Technology vs. economics
  - Technology vs. psychology
  - Technology vs. deep pocket

Complex Instruction Set Architecture (CISC)

Reduced Instruction Set Architecture (RISC)

# Maybe Review (from programming 101)?

P&H: Appendix B for further info



# Example C Program

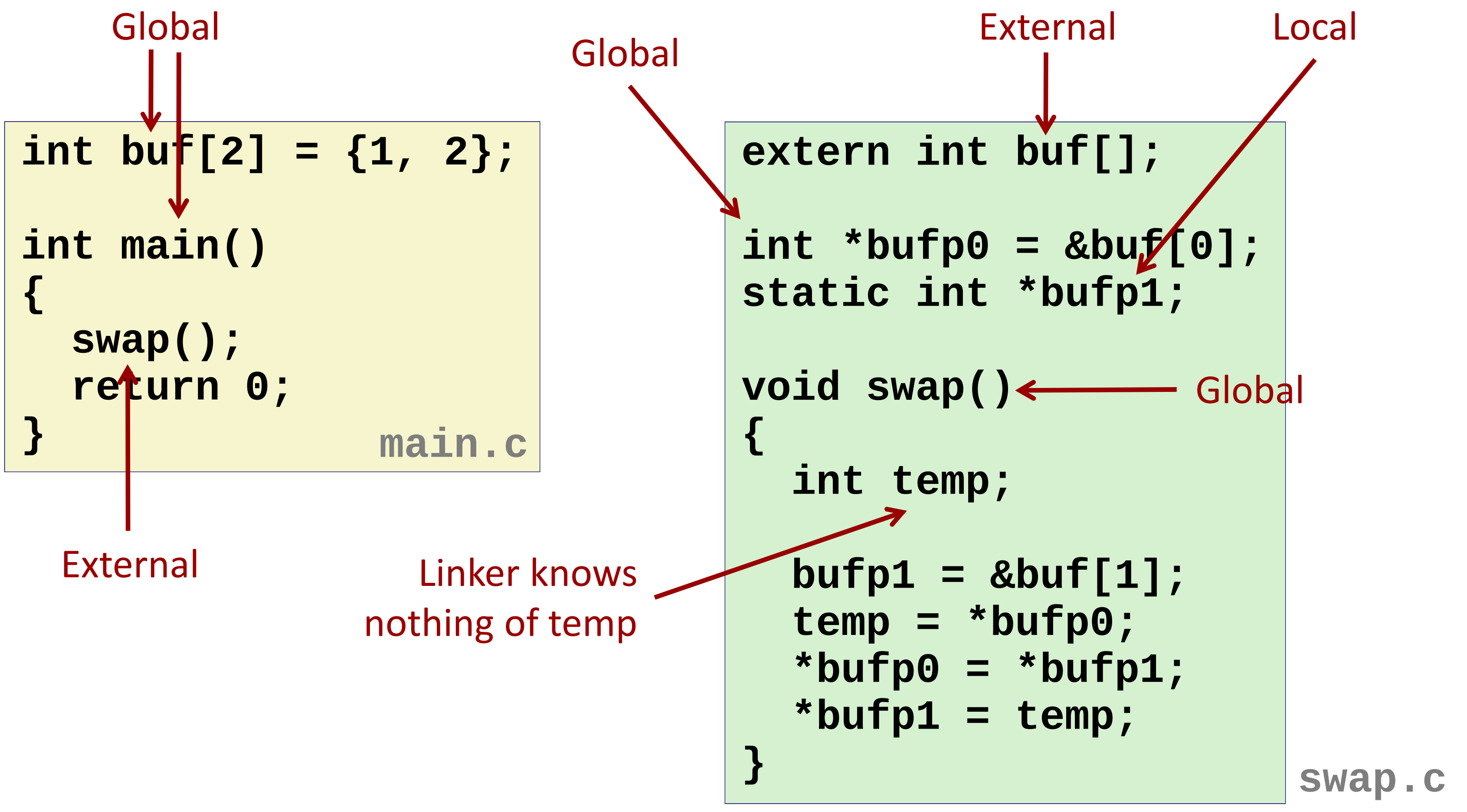
**main.c**

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

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

**swap.c**

```c
extern int buf[];

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

void swap()
{
  int temp;

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

# Resolving Symbols



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

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

main.c

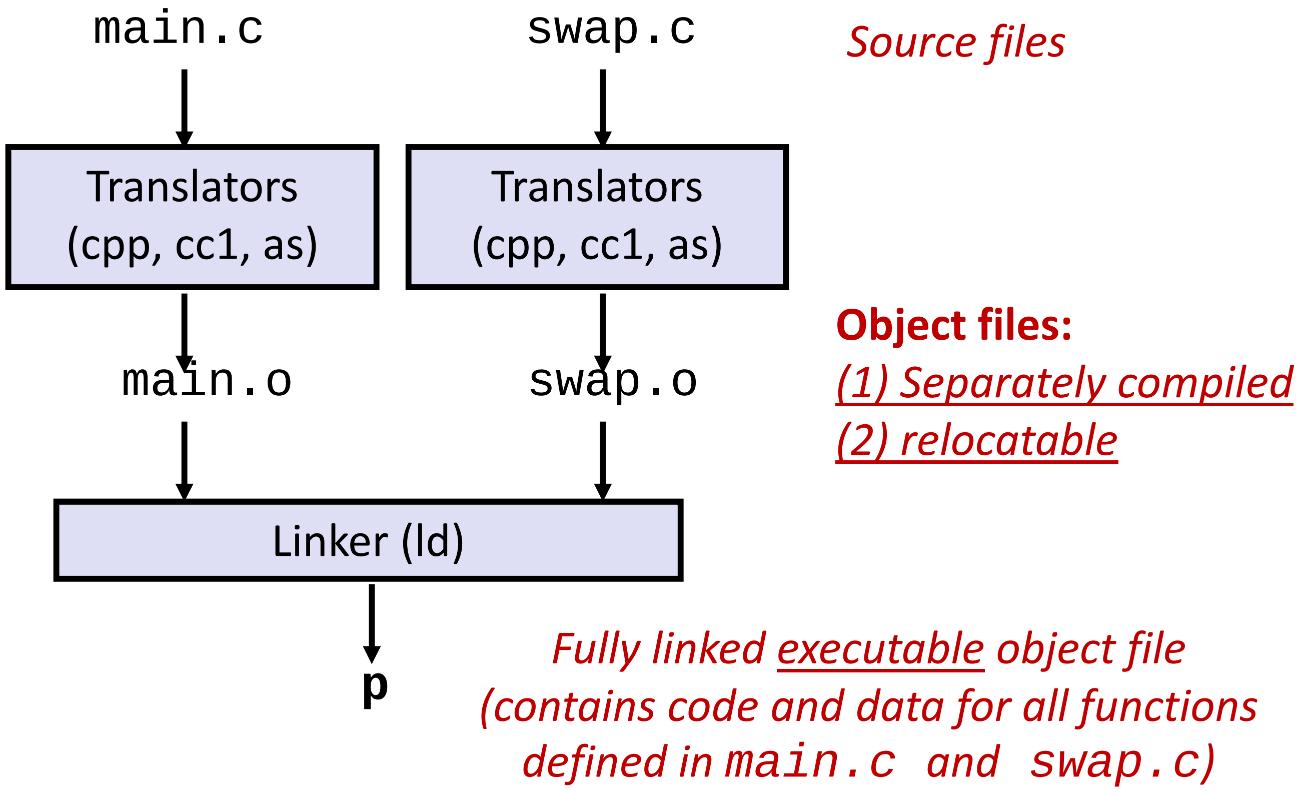
```
extern int buf[];

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

void swap()
{
  int temp;

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

swap.c

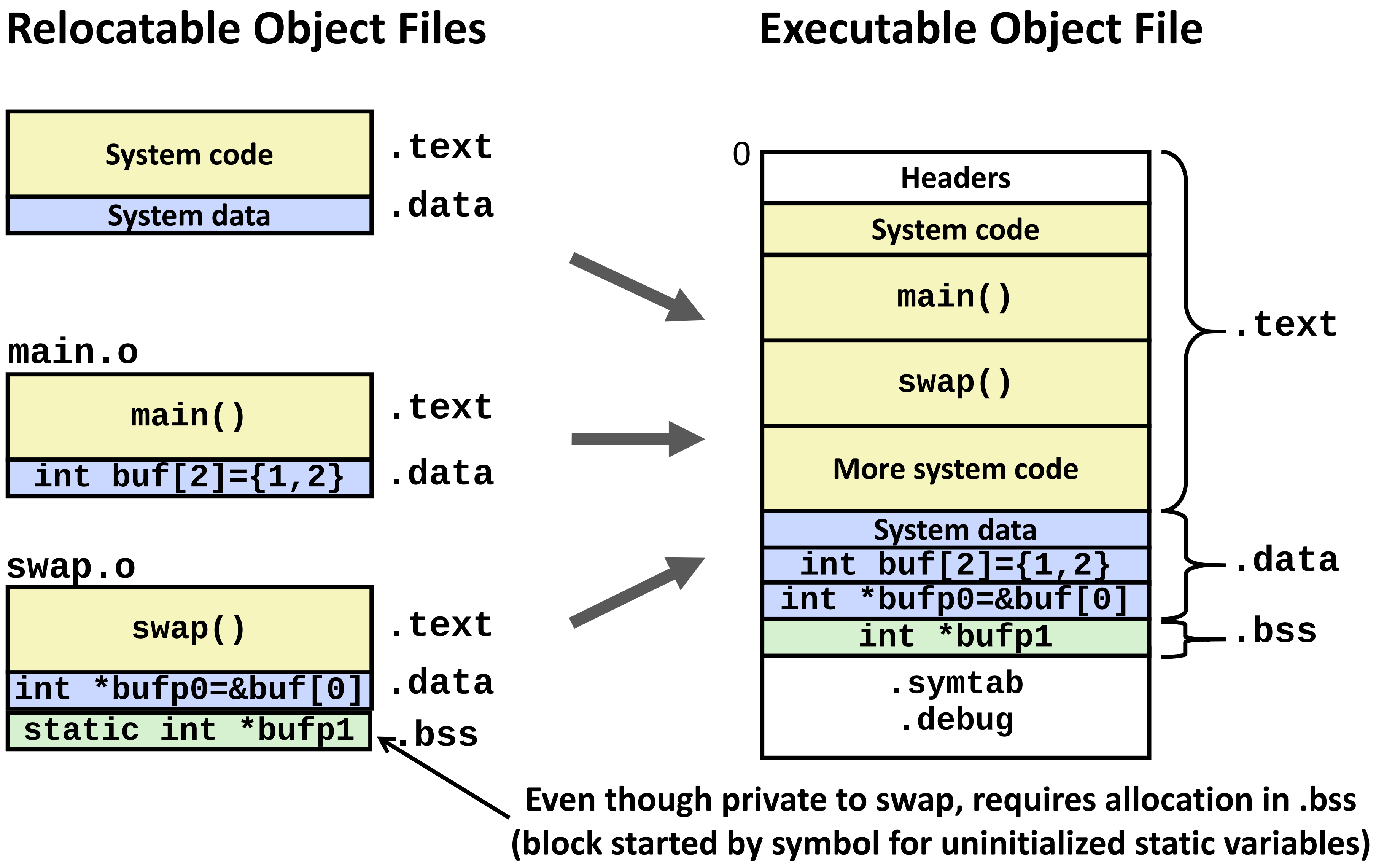
Information necessary to support linking

# Static Linking

- Programs are translated and linked using a *compiler driver*:
  - unix> *gcc -O2 -g -o p main.c swap.c*
  - unix> *./p*

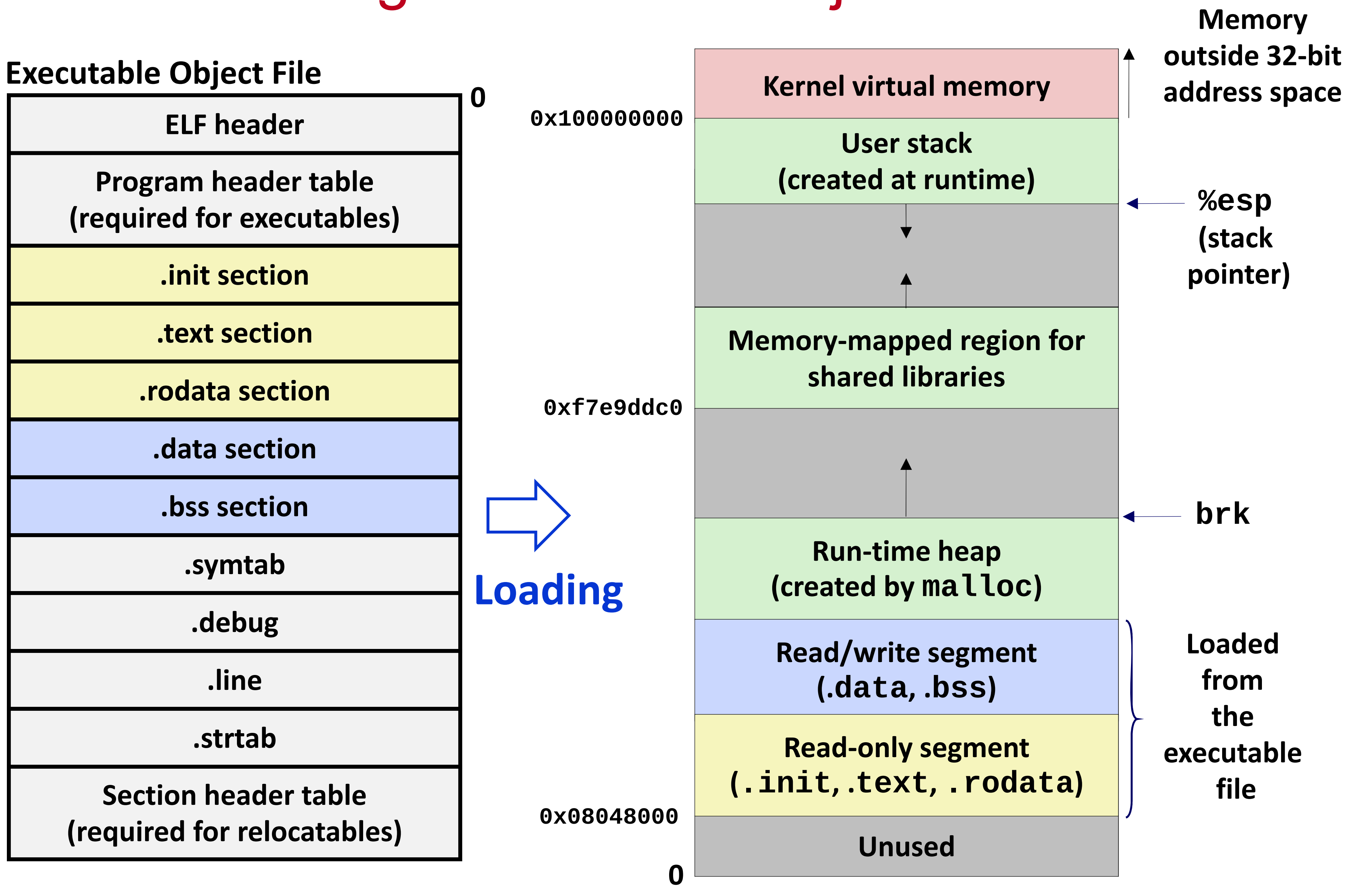

main.c
swap.c
*Source files*
Translators (cpp, cc1, as)
Translators (cpp, cc1, as)
main.o
swap.o
**Object files:**
*(1) Separately compiled*
*(2) relocatable*
Linker (ld)
**p**
*Fully linked executable object file (contains code and data for all functions defined in main.c and swap.c)*

# Relocating Code and Data



Even though private to swap, requires allocation in .bss
(block started by symbol for uninitialized static variables)

# Loading Executable Object Files



## What if you have to design an ISA?

## Open-Ended Design: extrapolation and anticipation

"a dependable base for a decade of customer planning and customer programming, and continuing laboratory development…" Typical life expectance 15~20 years, but…..

- "Asynchronous" operation of components
  - Abstract out exact time, performance etc to allow (1) changing technology and (2) relative speed of components
- Parameterization of storage capacity, multi CPU, multi I/O, etc
- Permit future extensions by "reserving" spare bits in instruction encoding
- Standard interfaces for expansion sub-systems

[Amdahl, Blaauw and Brooks, 1964]

## General Purpose

- General Purpose = effective support for "large and small, separate and mixed applications" in many domains (e.g., commercial, scientific, real-time….)

- How
  - Code-independent operation
    - No special interpretation of bit pattern in data

      e.g. ASCII character has no special significance.
    - Except where essential

      e.g., Integer, floating point, etc
  - Support full generality of logic manipulation on bit and data entities
  - Fine-grain memory addressability (down to small units of bits)

[Amdahl, Blaauw and Brooks, 1964]

## Inter-Model Compatibility

- **Strict program compatibility** = "a valid program whose logic will not depend implicitly upon time of execution and which runs upon configuration A, will also run on configuration B if the latter includes at least the required storage, at least the required I/O devices …."
- Invalid programs […] are not constrained to yield the same result
  - "invalid program" means a program that violates the architecture manual and not that it generates exceptions
  - exceptional conditions are part of the architecture
- By virtualization of logical structures and functions
- The King of Binary Compatibility: Intel x86, IBM 360
  - software base
  - performance scalability

[Amdahl, Blaauw and Brooks, 1964]

## Wrap-up: Terminologies

- Instruction Set Architecture (ISA)
  - The machine behavior as observable and controllable by the programmer
- Instruction Set
  - The set of commands understood by the computer
- Machine Code
  - A collection of instructions encoded in binary format
  - Directly consumable by the hardware
- Assembly Code
  - A collection of instructions expressed in “textual” format

    e.g. Add r1, r2, r3
  - Converted to machine code by an assembler
  - One-to-one correspondence with machine code

## Question?

*Announcements: ISA part.2 in the next lecture*

***Reading:*** ***P&H Ch 2 & optionally Appendix E (summary of all major ISAs)***

*Handouts: None*