System: Software: C & its architectural support-I

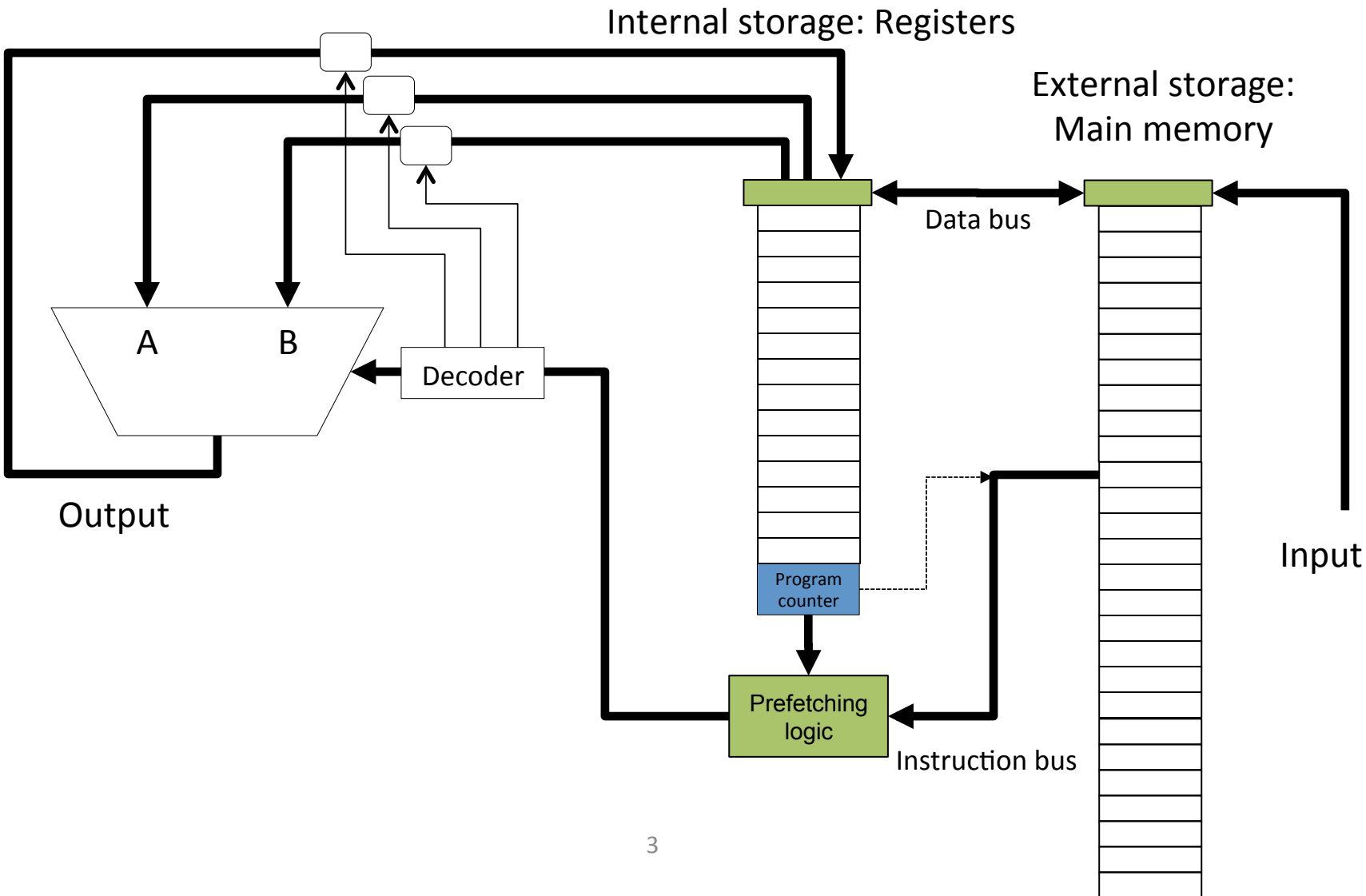
Lin Zhong

ELEC424, Fall 2014

Registers

Load-store architecture

Except load and store, instructions only operate on registers



Registers

- General purpose
  - R0 to R12
- Stack pointer (Will revisit)
  - R13 (MSP/PSP)
  - Main stack pointer: used by OS for exception handling
  - Process stack pointer: used by user programs
- Special registers

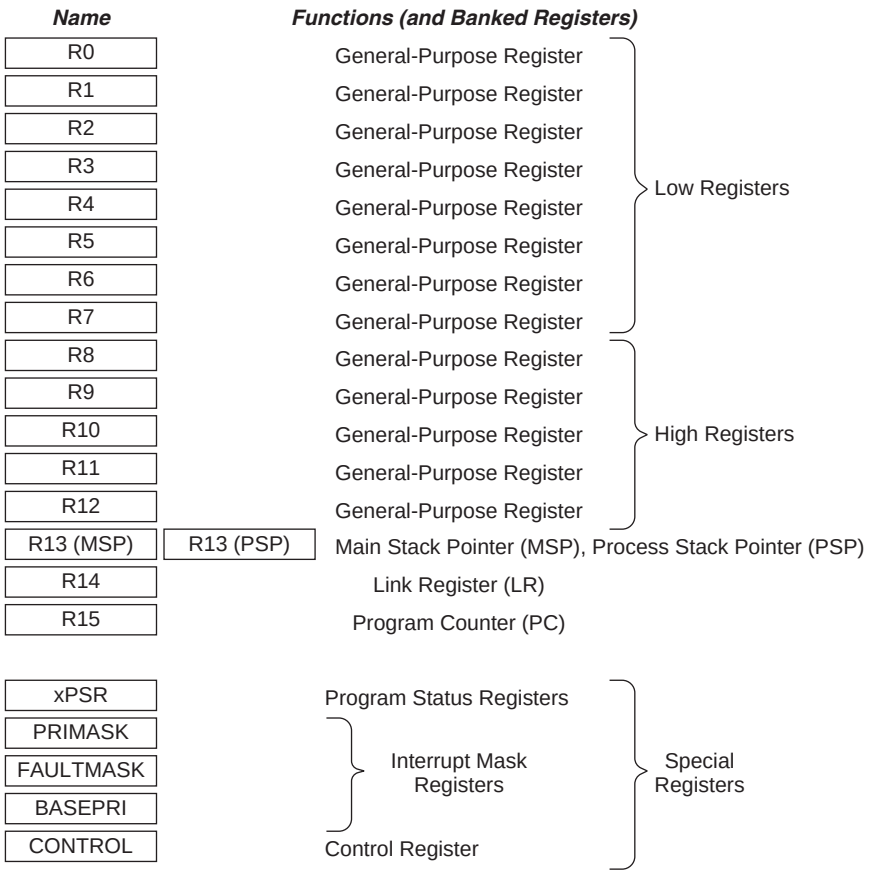


Figure 3.1 Registers in the Cortex-M3

Registers are the ***execution context*** of a processor
Registers define the ***state*** of a processor

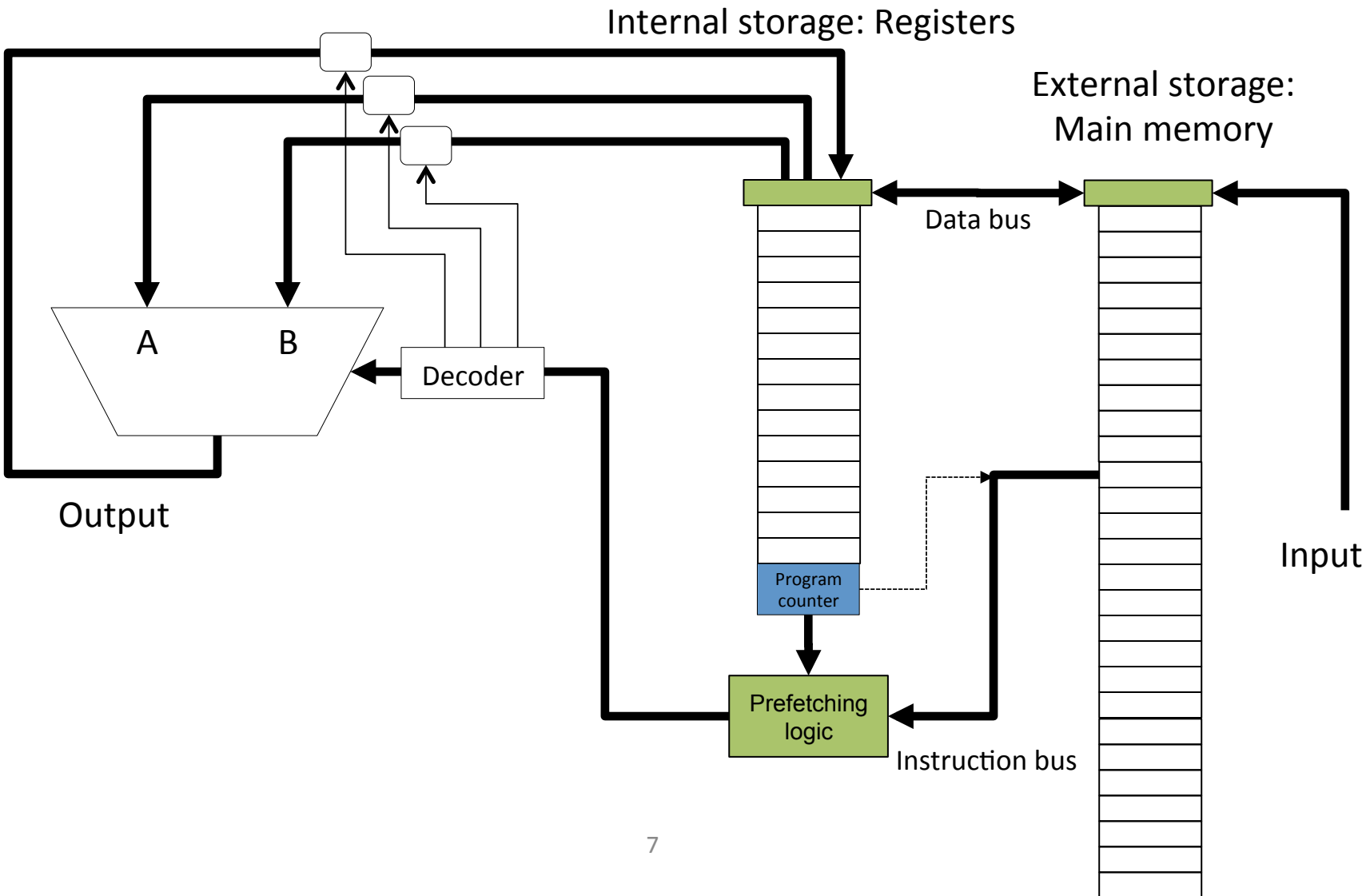
Special registers

Table 2.1 Registers and Their Functions

Register	Function
xPSR	Provide ALU flags (zero flag, carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and HardFault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

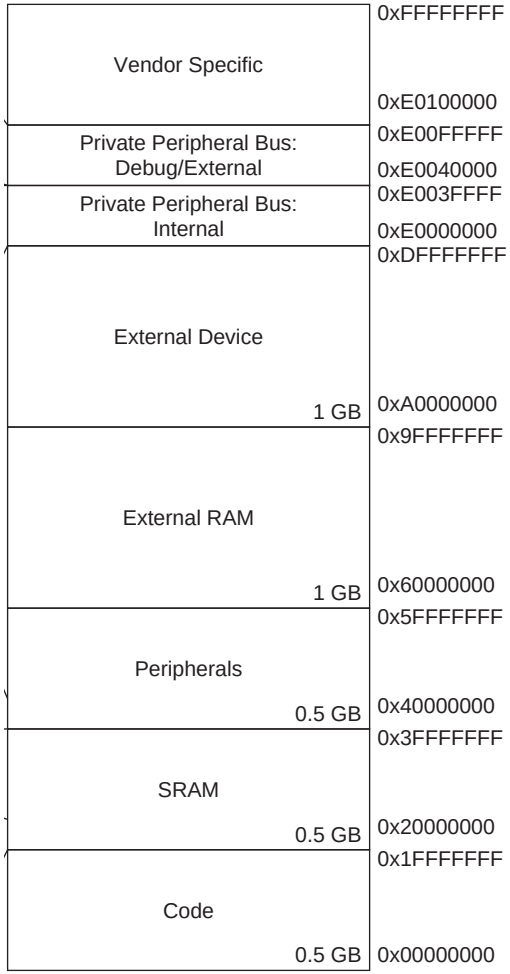
Who determines the use of (general-purpose registers)?

Compiler: map variables to registers (register allocation)



Memory

Cortex-M3: Predefined memory map



- Bufferable (for writes),
- Cacheable
- Executable
- Sharable

What happens after reset?

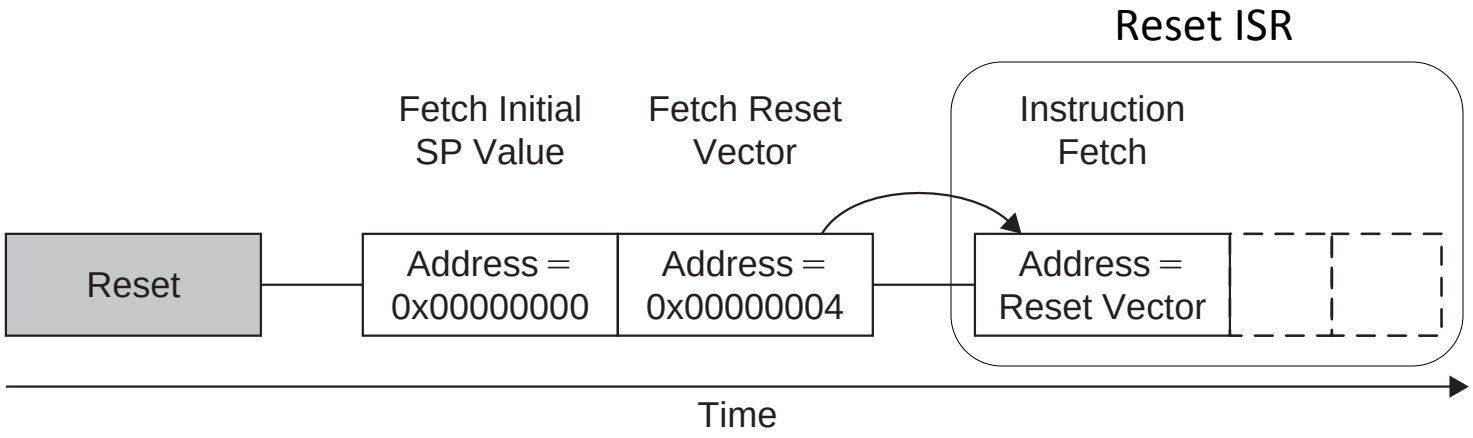


Figure 3.17 Reset Sequence

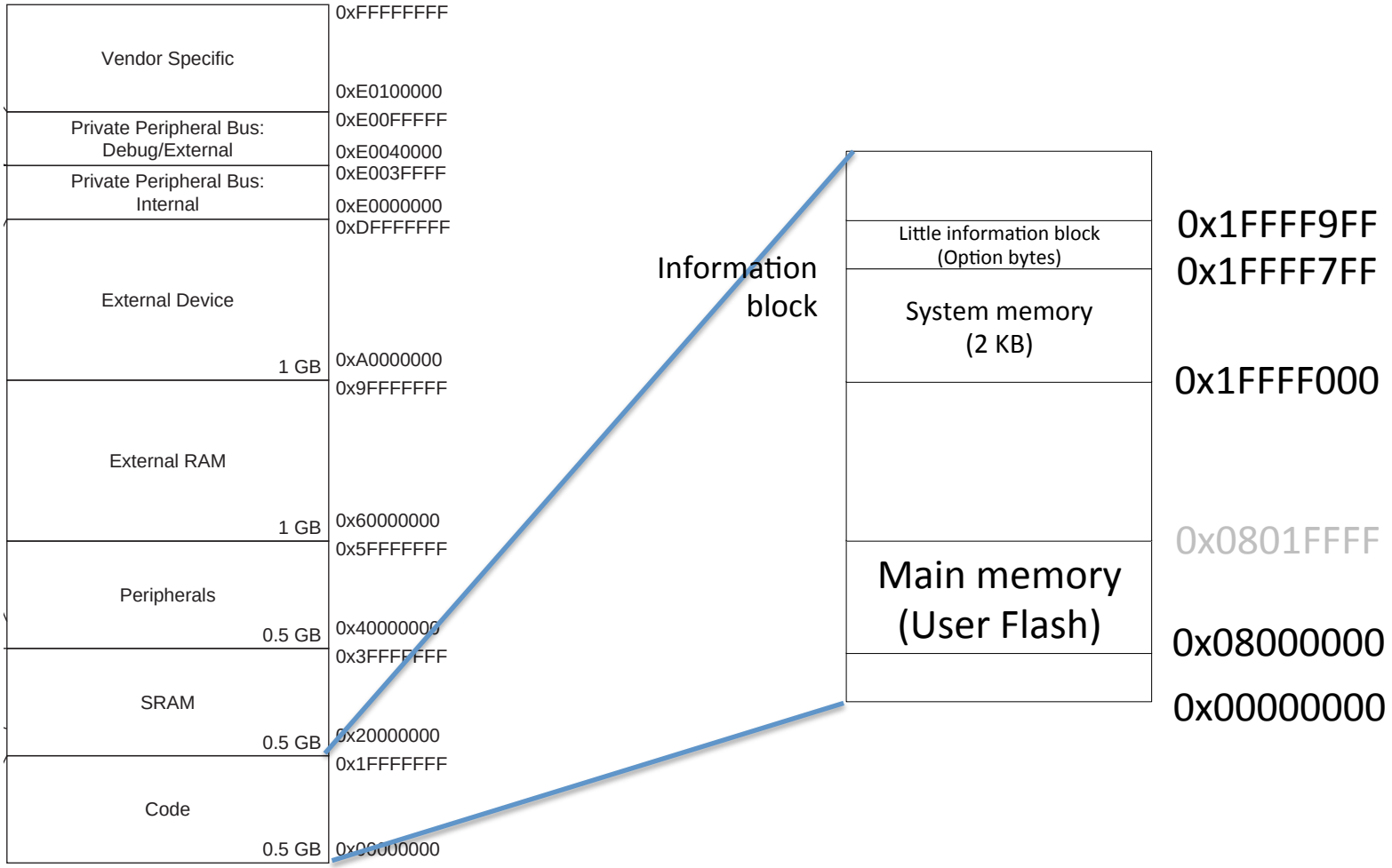
Homework I (Individual, not team)

- How does the STM32 microcontroller's booting process respect Cortex-M3's yet provide more flexibility?

- Read the proper parts of the reference manual

- Due Thursday lecture (hand in before class)

Cortex-M3 vs. SMT32



Why doesn't it place Reset ISR at 0x0000004?

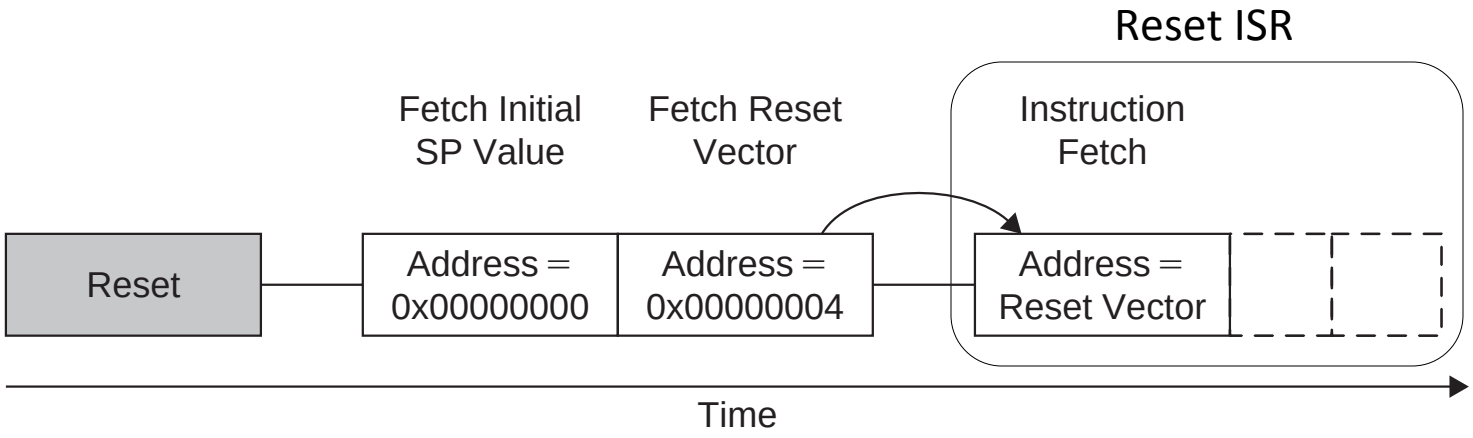


Figure 3.17 Reset Sequence

System Design Principle III: Indirection

- "All problem in computer science can be solved by another layer of indirection"
  - David Wheeler, Quoted by Butler Lampson in his Turing Award acceptance lecture
  - http://en.wikipedia.org/wiki/Indirection

- Address 0x00000004 provides indirection for Reset ISR
- P.O. Box
- Alias

More examples

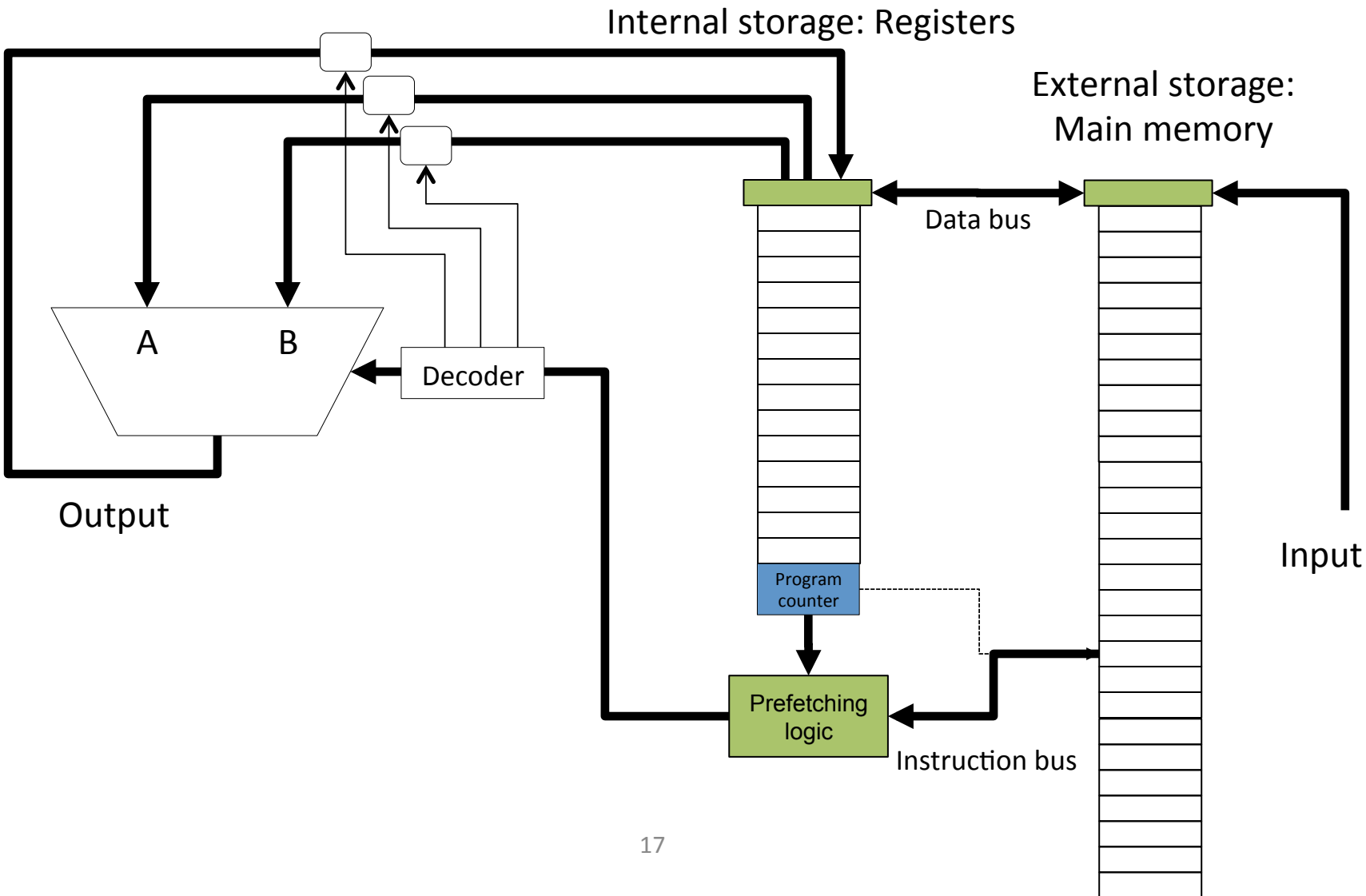
- Default boot memory location
  - Cortex M3 vs. STM32f10x

Abstraction vs. Indirection

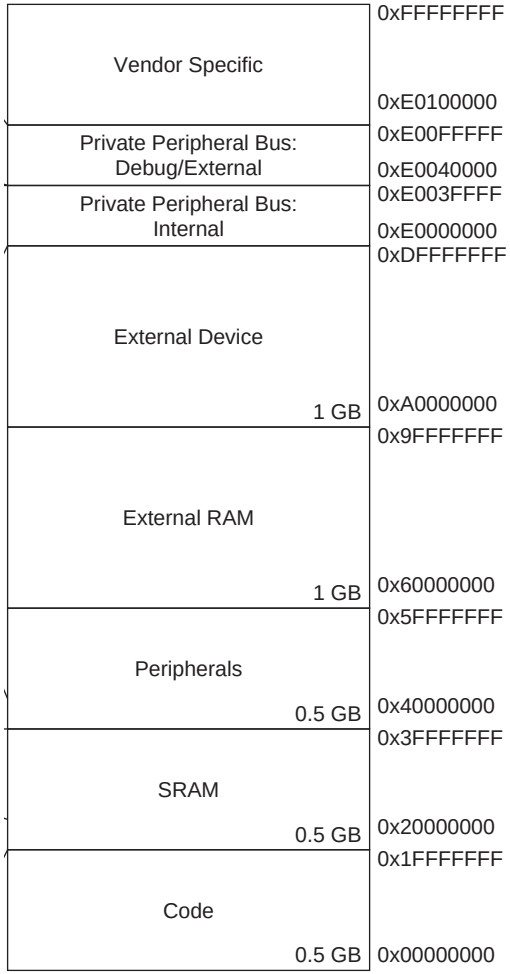
- Separation of interface from implementation
- Alias of interface

Where we left last time

Except load and store, instructions only operate on registers



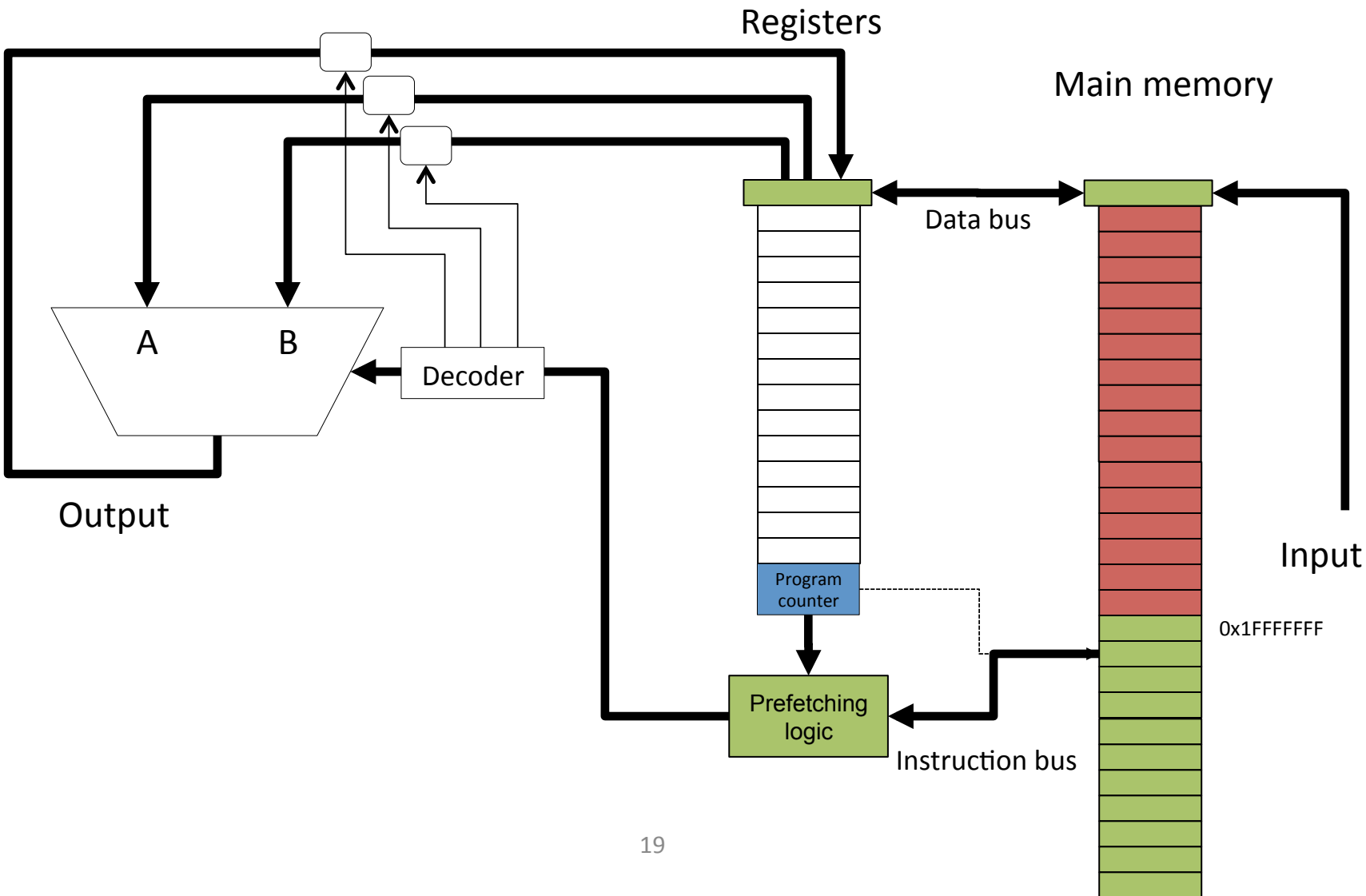
Cortex-M3: Predefined memory map



- Bufferable (for writes),
- Cacheable
- Executable
- Sharable

SRAM vs. Flash

Flash is usually used for code; SRAM for data

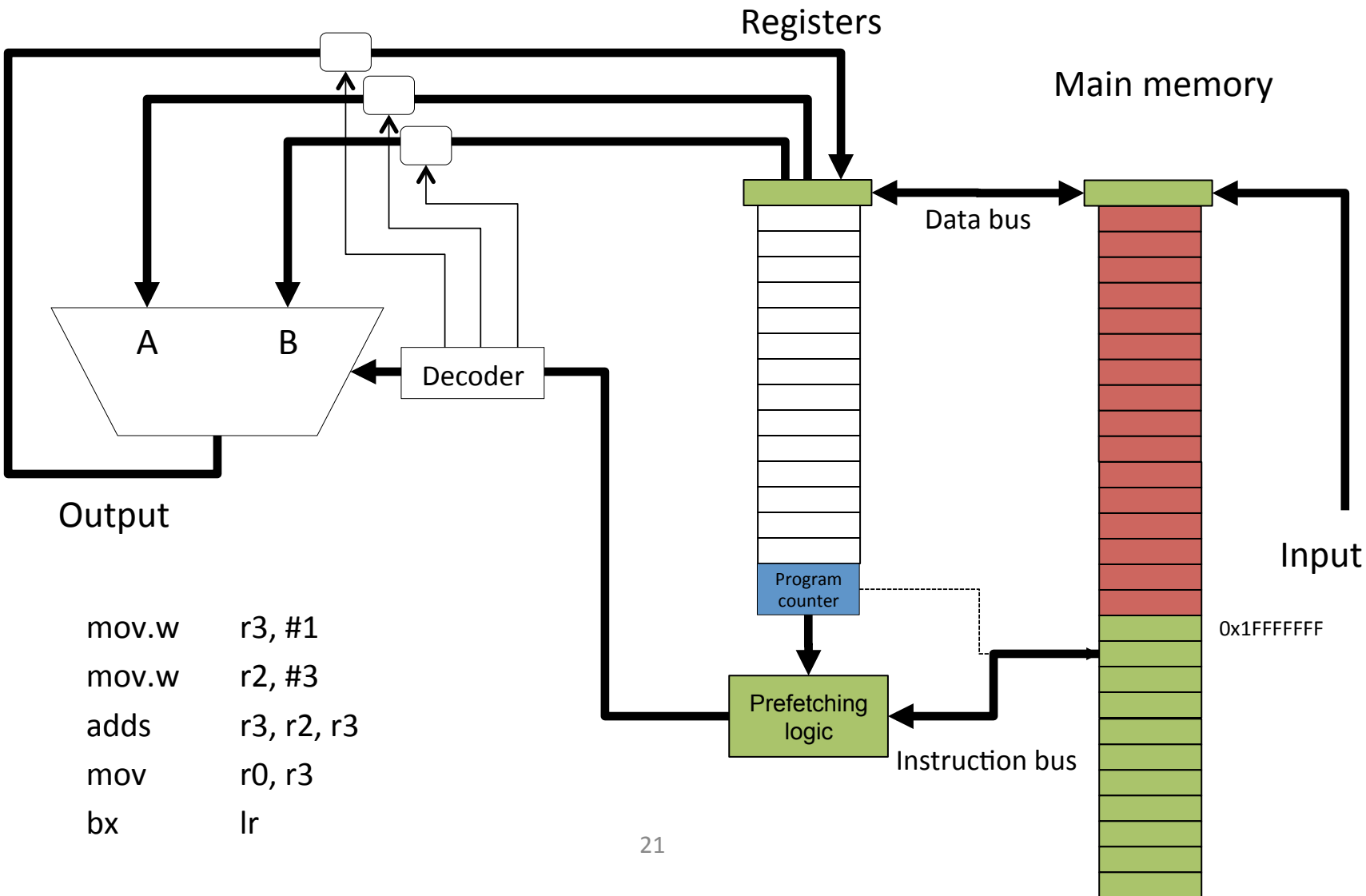


Hand-optimized code

```
int main()
{
  int a=1, b=3;
  return a+b;
}
```

```
        mov.w r3, #1
        mov.w r2, #3
        adds  r3, r2, r3
        mov   r0, r3
        bx    lr
```

Only registers are used



Non-optimized code from compiler

gcc –O0

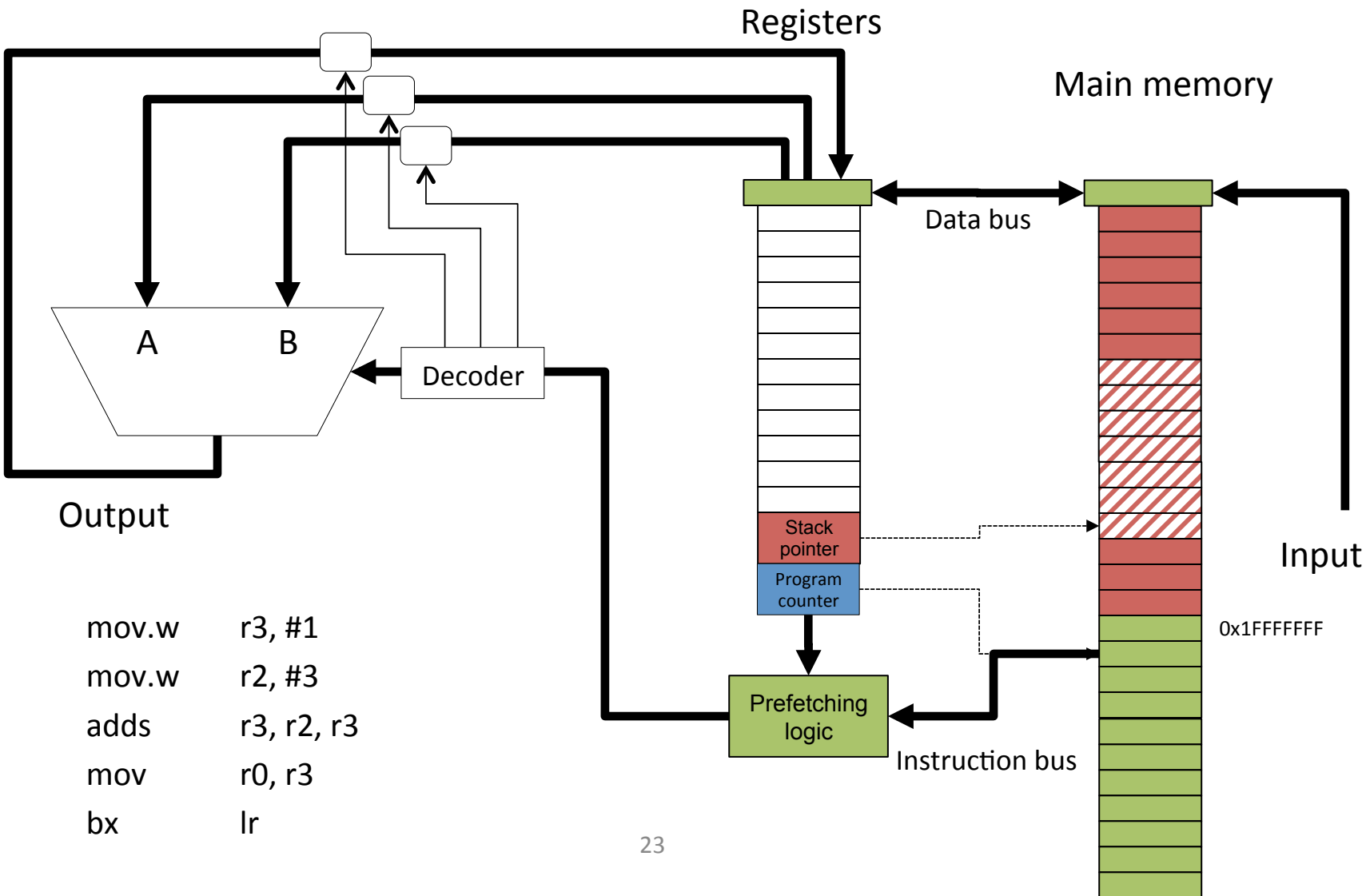
```
int main()
{
  int a=1, b=3;
  return a+b;
}
```

```
0:   b480       push   {r7}
2:   b083       sub    sp, #12
4:   af00       add    r7, sp, #0
6:   f04f 0301  mov.w  r3, #1
a:   607b       str    r3, [r7, #4]
c:   f04f 0303  mov.w  r3, #3
10:  603b       str    r3, [r7, #0]
12:  687a       ldr    r2, [r7, #4]
14:  683b       ldr    r3, [r7, #0]
16:  18d3       adds   r3, r2, r3
18:  4618       mov    r0, r3
1a:  f107 070c  add.w  r7, r7, #12
1e:  46bd       mov    sp, r7
20:  bc80       pop    {r7}
22:  4770       bx     lr
```

Compiled without optimization;
arm-linux-gnueabi-objdump –d main.o

Stack: a way to use the memory

Architectural support: stack pointer



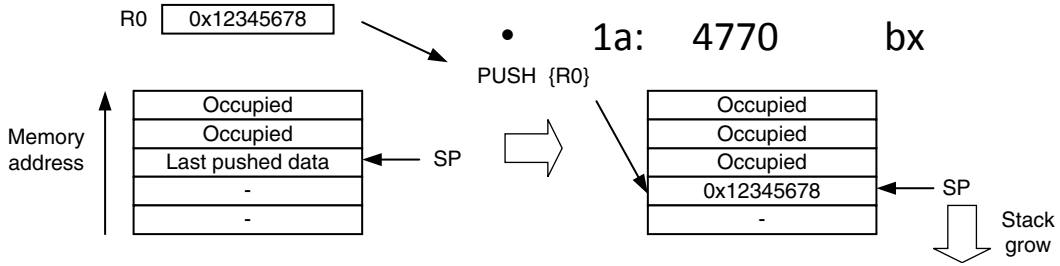

```
mov.w   r3, #1
mov.w   r2, #3
adds    r3, r2, r3
mov     r0, r3
bx      lr
```

PUSH and POP instructions to use stack efficiently

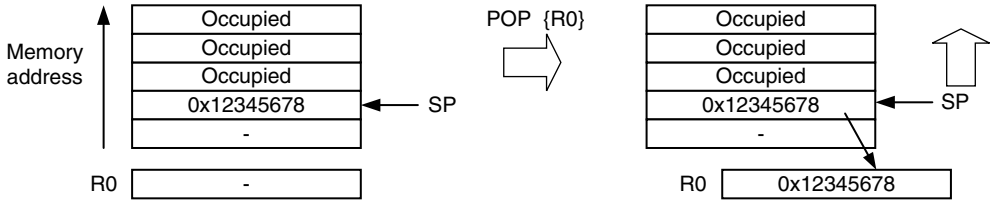
Hand-coded to use stack instructions

- int main()
- {
- int a=1, b=3;
- return a+b;
- }

```
0:   f04f 0301   mov.w   r3, #1
4:   607b        push    r3
6:   f04f 0303   mov.w   r3, #3
10:  603b        push    r3
12:  687a        pop     r2
14:  683b        pop     r3
16:  18d3        adds    r3, r2, r3
18:  4618        mov     r0, r3
1a:  4770        bx      lr
```



Cortex-M3's PUSH and POP



Stack

- You do not HAVE to use PUSH and POP to use stack
  - Stack pointer (SP) as the reference
  - Memory locations can be referred as relative to SP
- PUSH and POP allow Last-In-First-Out access
  - Will revisit for context switching

- PUSH and POP use dedicated hardware
  - faster and more efficient
  - “PUSH r3” vs. “str r3, [r7, #4]”
  - No need to calculate the memory address when PUSH

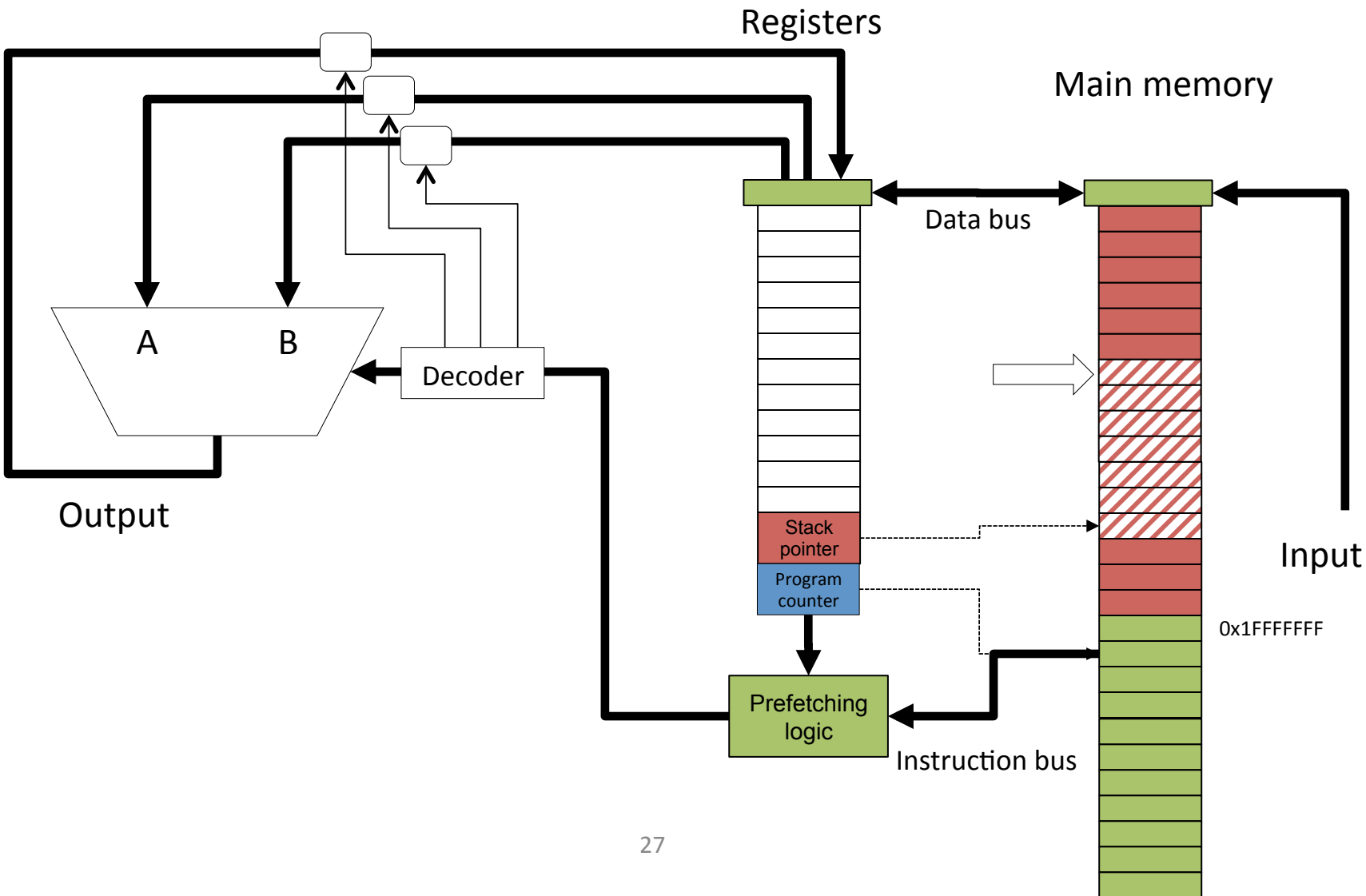
Stack:

A continuous segment of memory vs. FILO data structure

- "Stack" is a loaded word

- In the most primitive sense, it is a continuous segment of memory dedicated to a program
  - This memory can be accessed with LOARD and STORE instructions, given the stack pointer
  - Stack pointer points to one end of this segment ("top" of the stack)
- When you access this segment using PUSH and POP instructions, it behaves as the stack data structure (FILO)

How is the stack bottom determined?



What happens after reset?

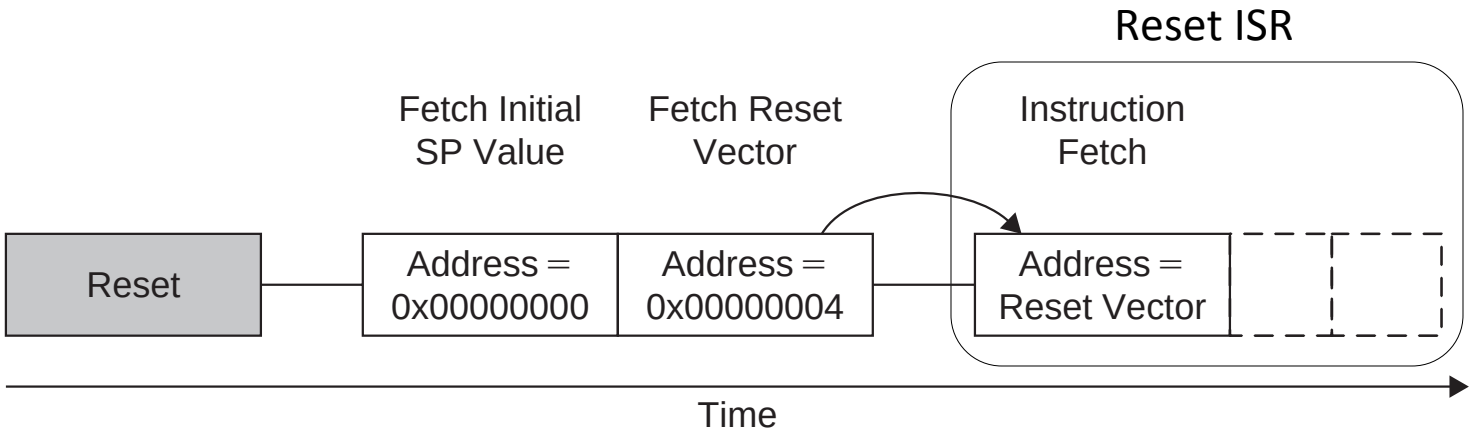


Figure 3.17 Reset Sequence

SP is set first thing after reset

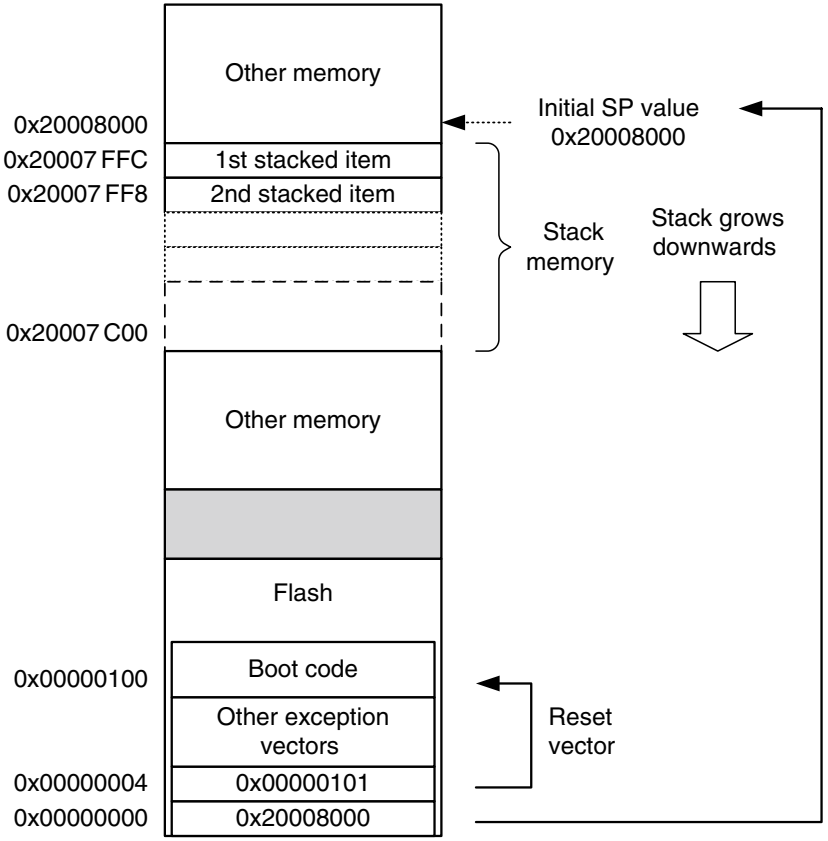
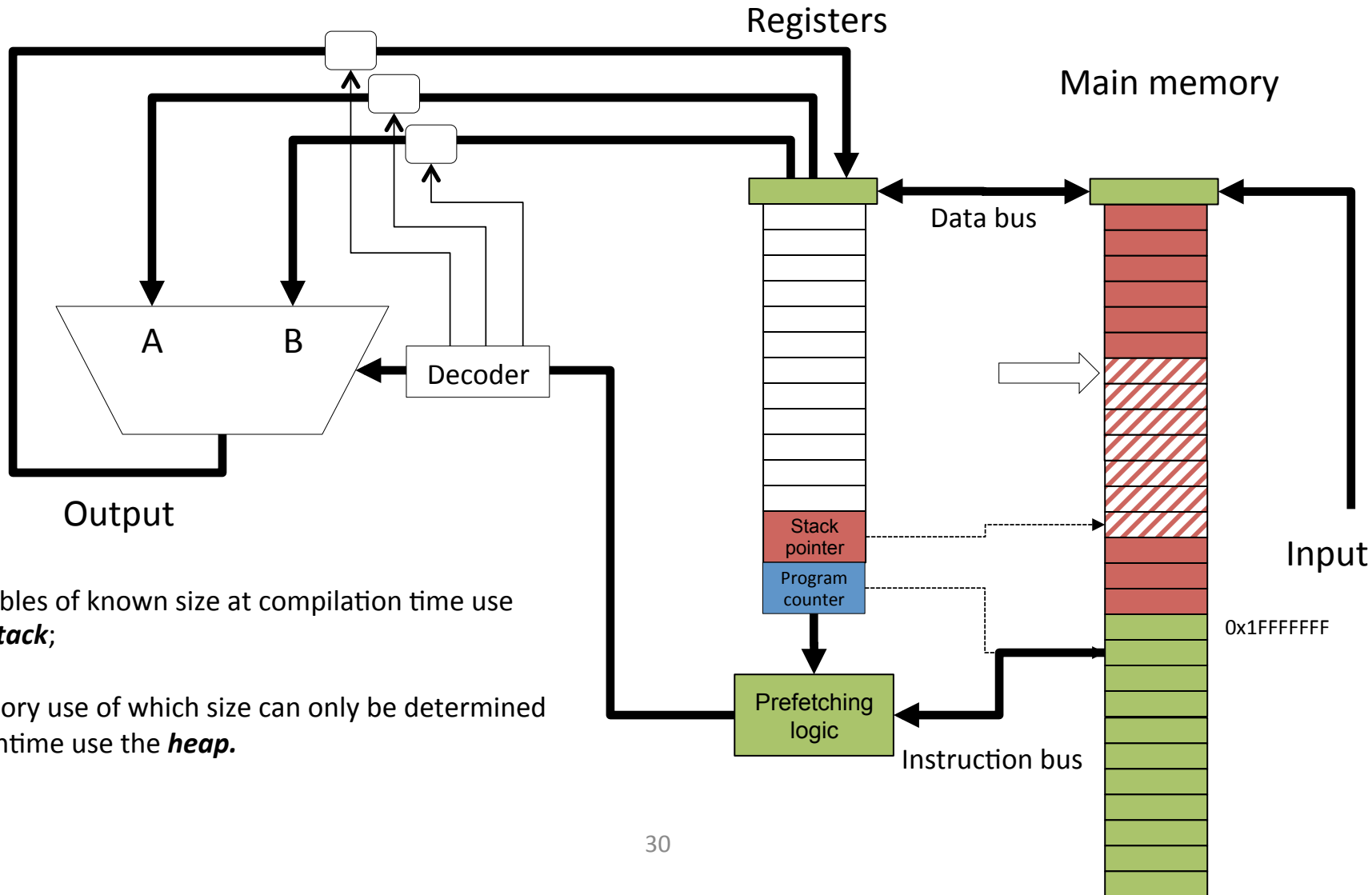


FIGURE 3.19

Initial Stack Pointer Value and Initial Program Counter Value Example.

Who determines stack use?

Compiler: Map “variables” to stack locations



Variables of known size at compilation time use the ***stack***;

Memory use of which size can only be determined at runtime use the ***heap.***