CSE 331: Microprocessor Interfacing and Embedded Systems

Lecture 3: ARM ISA (Assembly Language)

CISC – Design Architecture

- **CISC – Complex Instruction Set Computer**
- History:
- Prior to 1970 – RAM's were slow and expensive
- Programs used to be written in High Level Languages (HLL) – user friendly
- Compiler converts these HLLs to low level language for hardware.
- As the HLLs were getting more complex, designing compiler was also becoming difficult for complex instructions

CISC – Design Architecture

- There was a need to bridge the gap between HLLs and computer architecture
- Compact and simple low level instructions
- These instructions takes less space in memory also compiler design became simpler
- Example compact instruction (CISC computation):
  - Multiply 5*8
  - Done with single command
  - First numbers from the mem loc are fetched and loaded to 2 separate registers
  - ALU performs the multiplication
  - Then the result is stored in the memory

CISC – Design Architecture

- Now the hardware underneath the program had to do take the burden of execution
- This made the hardware more complex
- As time went on, processors started supporting more diverse instructions
  - Multply
  - Clear
  - Load
  - Add
  - Subtract
  - Branch
  - Move
  - …

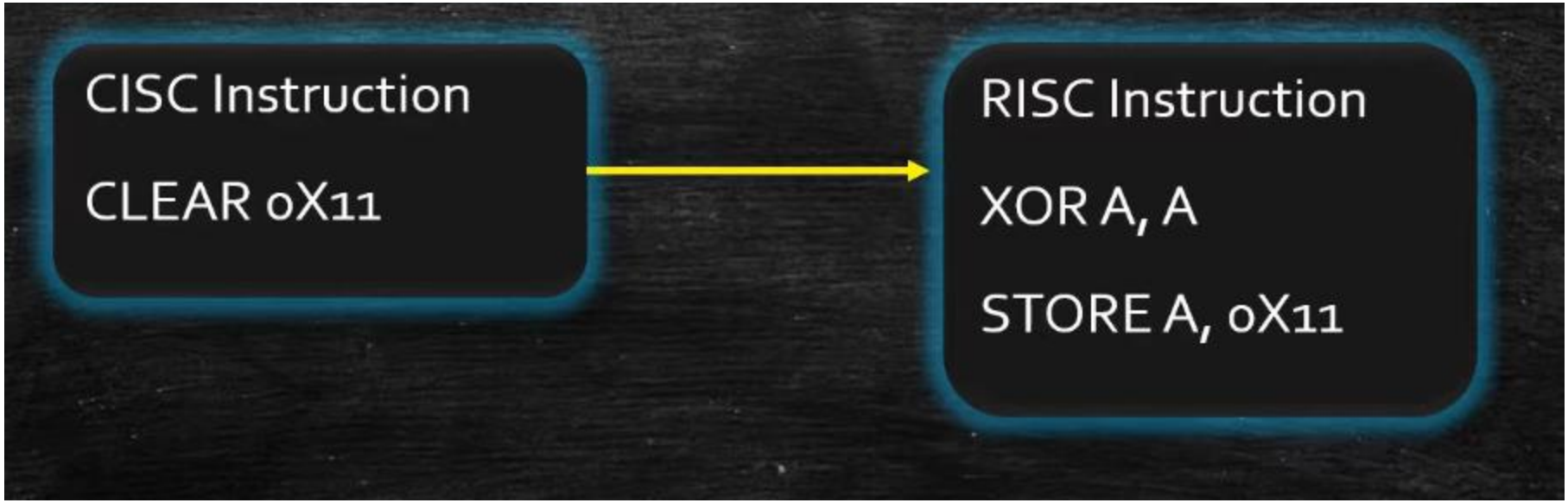
  - So the processor design started getting more complicated
  - More silicone/transistors were needed for supporting this increased number of instructions
- Large and expensive processor
- Increased Power consumption

CISC – Design Architecture

- It was noticed that only 20% of the instructions were being used 80% of the time
- Many complex instructions were not being used

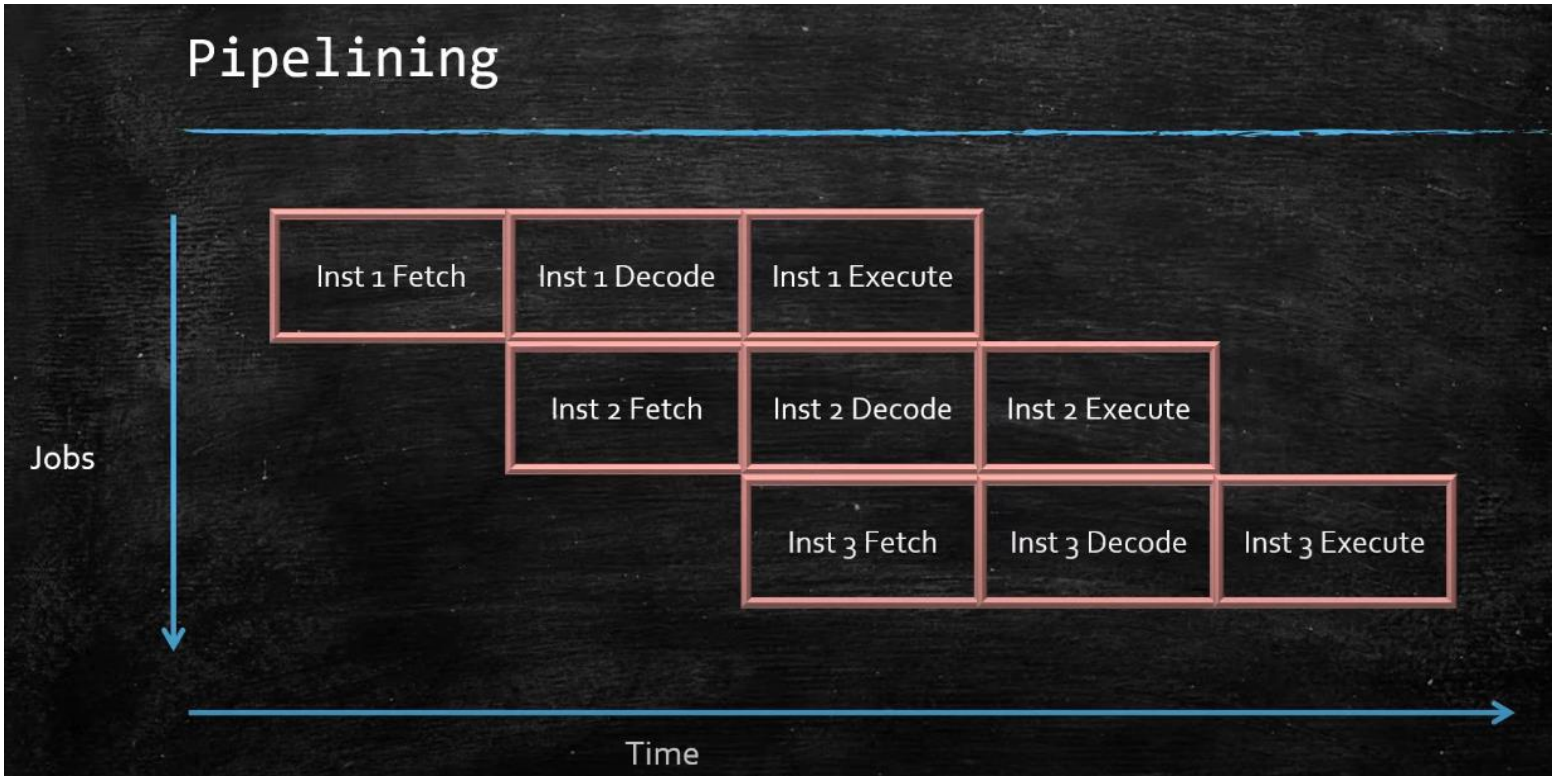
- It made more sense to only have only simple and most commonly used instructions
- That is how RISC – Reduced Instruction Set Computer was conceptualized
- Low number of instructions – Less complex hardware
- However this doesnot mean RISC has less capability.
- RISC used these reduced simpler set of instructions to perform all the complex tasks

RISC vs CISC instruction comparisom



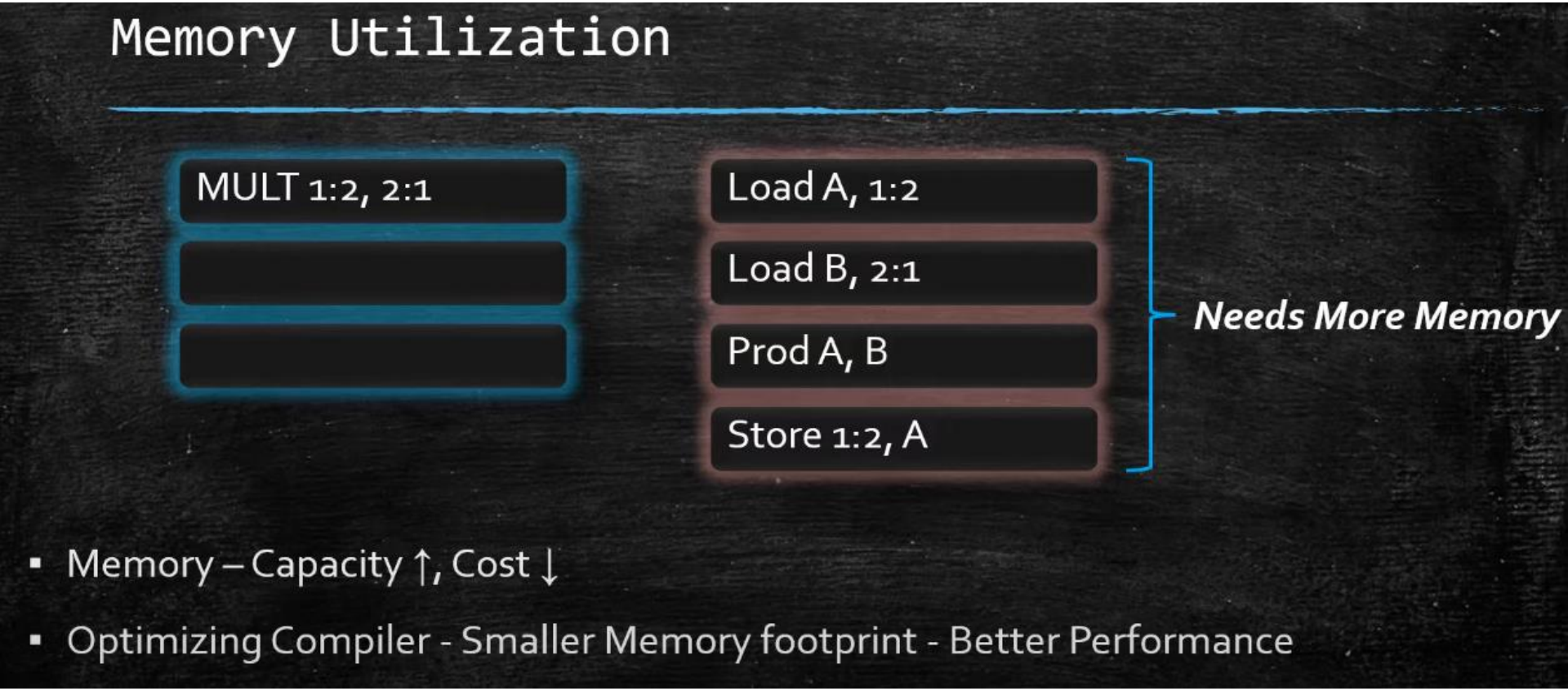
- CISC needed only 1 Instruction where RISC needs more than 1 instructions
- However, RISC instructions are simpler and of same length
- Typically 1 instruction = 1 clock cycle
- Faster execution speed

RISC vs CISC instruction comparisom



- Pipelining allows to execute multiple instruction at the same time in parallel.
- Thus making things faster.

RISC vs CISC instruction comparisom



RISC vs CISC instruction comparisom

- CISC Needed more number of transistors to support comlpex instructions
- In RISC it needs less transistors due to simplicity of instructions
- A lot of the transistors were saved
- These were used to increase number of
  - Registers
  - Cache
  - Other units

- More registers mean, processor doesnot need to access the main memory frequently (contrary to CISC)

RISC vs CISC

- **Full forms: R**educed/**C**omplex **I**nstruction **S**et **C**omputers
- **Instruction size:** Fixed vs Variable
- **Instruction Fetch time:** same for all instruction vs Vary with instructions
- **Instruction Set:** Small and Simple vs Large and Complex
- **Addressing Mode:** Less modes as instructions are on Registers. Vs More modes of complex instructions as they can be on both reg and memory as well.
- **Number of Reg:** Many vs Few
- **Compiler Design**: Simple vs Complex
- **Program size**: long vs small
- **Number of Operands:** Fixed(mainly in reg) vs Variable(mem and reg)
- **Control Unit:** Hardware control vs Micro Program Control(as hw control will become complex for more instructions)
- **Exectution Speed:** Fast vs Slower
- **Pipelining**: More effective vs Less Efective

CISC & RISC

- CISC: complex instruction set computer
- original CPUs very simple
- poorly suited to evolving high level languages
- extended with new instructions
  - e.g. function calls, non-linear data structures, array bound checking, string manipulation
- implemented in *microcode*
  - sub-machine code for configuring CPU sub-components
  - 1 machine code == > 1 microcode
- e.g. Intel X86 architecture

CISC & RISC

- CISC introduced integrated combinations of features
  - e.g. multiple address modes, conditional execution
  - required additional circuitry/power
  - multiple memory cycles per instruction
  - over elaborate/engineered
    - many feature combinations not often used in practise
- could lead to loss of performance
- better to use combinations of simple instructions

CISC & RISC

- RISC: reduced instruction set computer
- return to simpler design
- general purpose instructions with small number of common features
  - less circuitry/power
  - execute in single cycle
- code size grew
- performance improved
- e.g. ARM, MIPS

ARM

- Advanced RISC Machines
- UK company
  - originally Acorn
  - made best selling BBC Microcomputer in 1980s
- world leading niche processors
- 2015 market share:
  - 85% of apps processors
  - 65% of computer peripherals,
  - 90% of hard-disk and SSD
  - 95% of automotive apps processors

ARM

- 32 bit RISC processor architecture family
- many variants:
  - A (Application), M (Microcontroller), R (Real Time), X (High Performance)
- ARM does not make chips
- licences the *IP core* (intellectual property) logic design to chip manufacturers

Books

- general reference :
  - W. Hohl, *ARM Assembly Language*, CRC Press, 2009
  - ARM7TDMI
  - NB not Linux format
- for Raspberry Pi:
  - B. Smith, *Raspberry Pi Assembly Language Raspbian Hands On Guide (2nd Ed)*, 2014

References

ARM Cortex A processor:

https://silver.arm.com/download/Software/BX100-DA-98001-r0p0-01rel3/DEN0013D_cortex_a_series_PG.pdf General Purpose IO:

https://www.raspberrypi.org/documentation/hardwar_e/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf

Memory

Memory is arranged as a series of "locations"

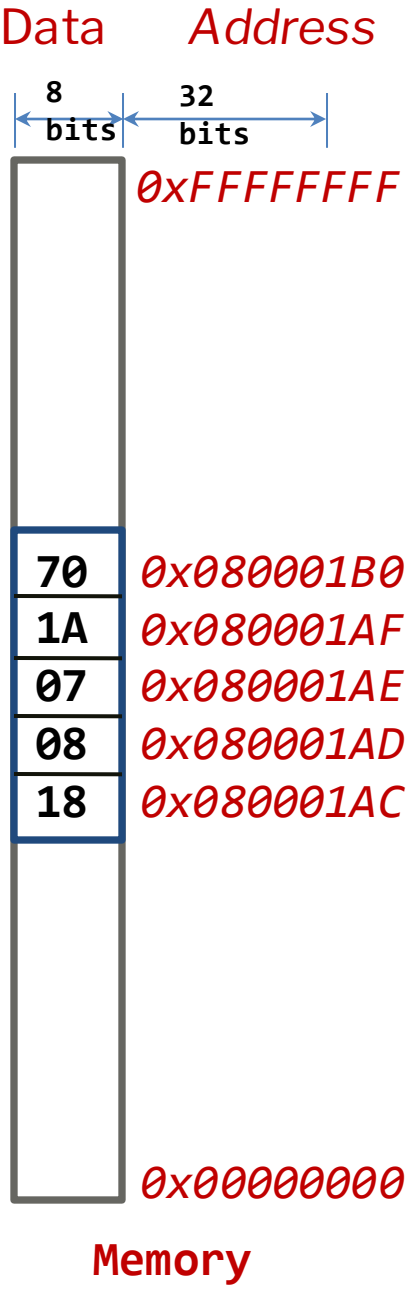
- Each location has a unique "address"
- Each location holds a byte (***byte-addressable***)
- e.g. the memory location at address `0x080001B0` contains the byte value `0x70`, i.e., 112

The number of locations in memory is limited

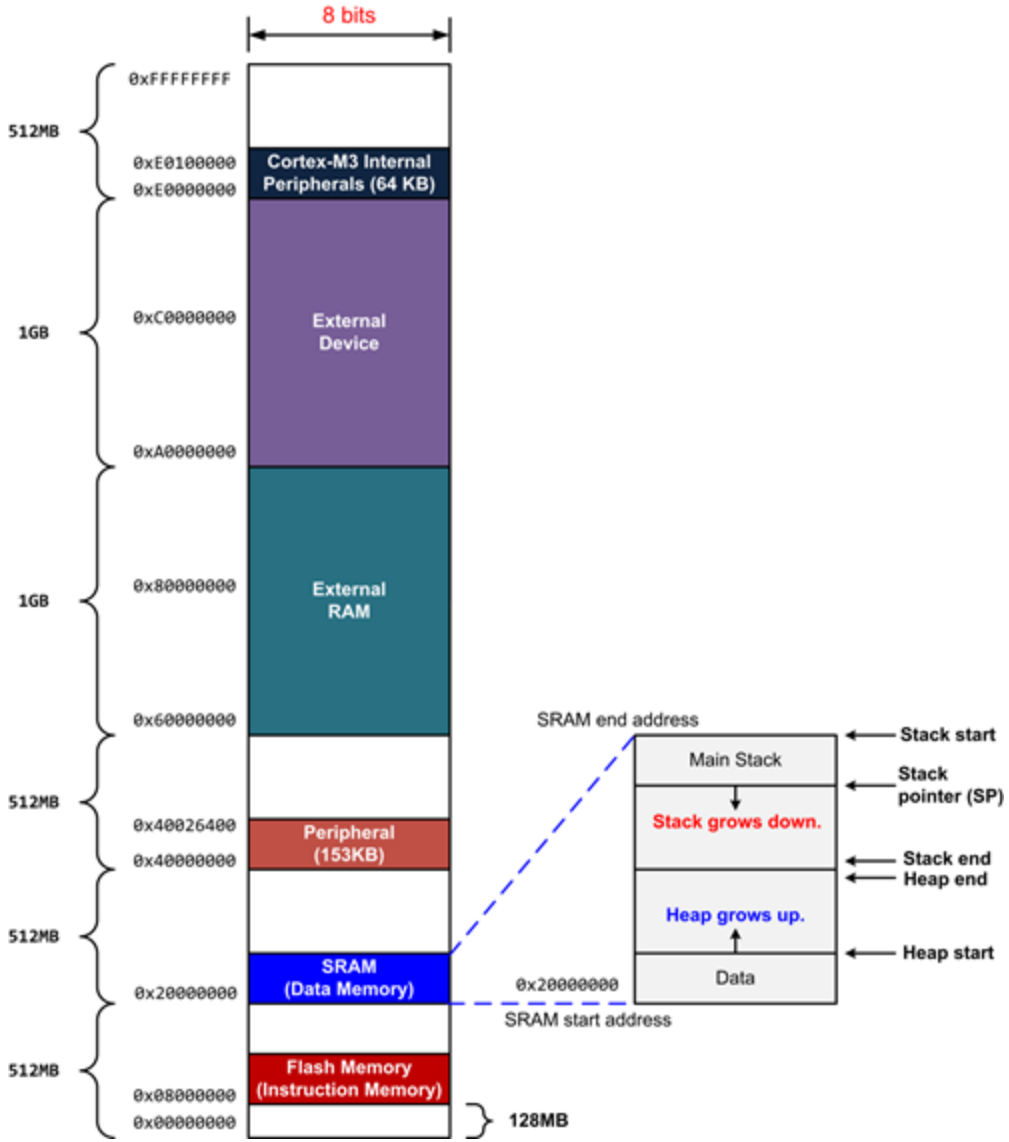
- e.g. 4 GB of RAM
- 1 Gigabyte (GB) = 2^{30} bytes
- 2^{32} locations ⍰ 4,294,967,296 locations!

Values stored at each location can represent either program data or program instructions

- e.g. the value `0x70` might be the code used to tell the processor to add two values together



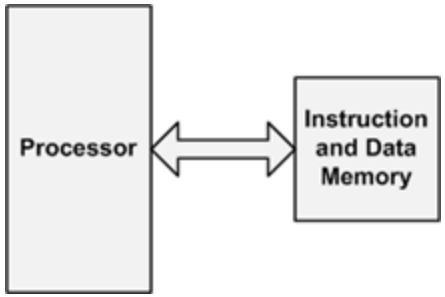
ARM M4 Memory Map



Computer Architecture

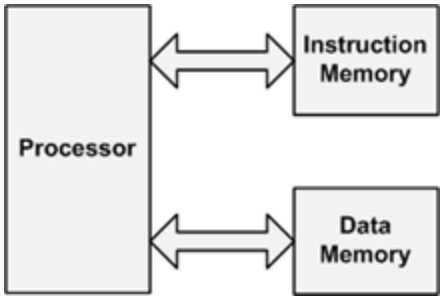
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

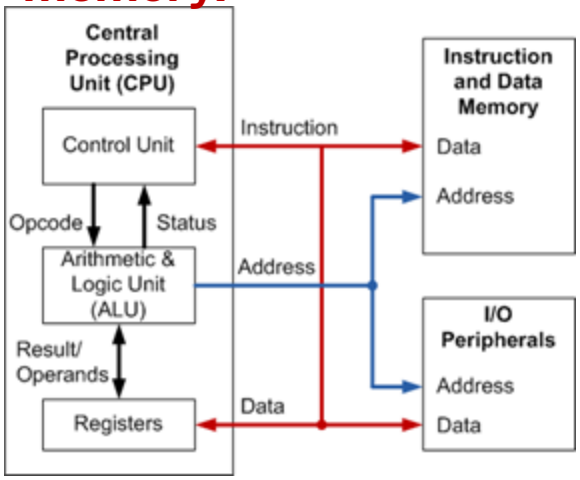
Data and instructions are stored into separate memories.



Computer Architecture

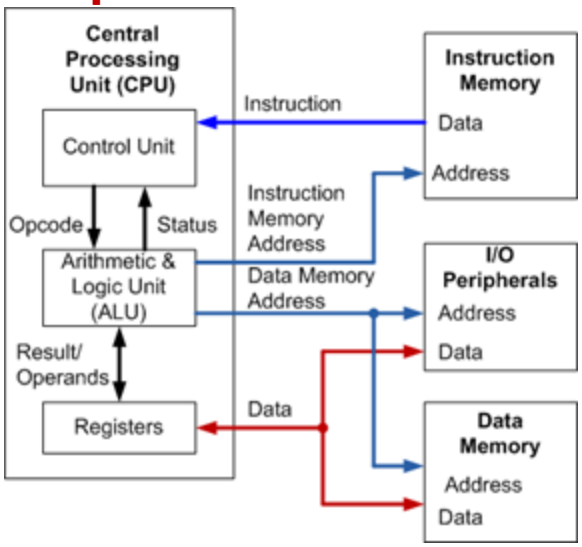
Von-Neumann

Instructions and data are stored in the same memory.

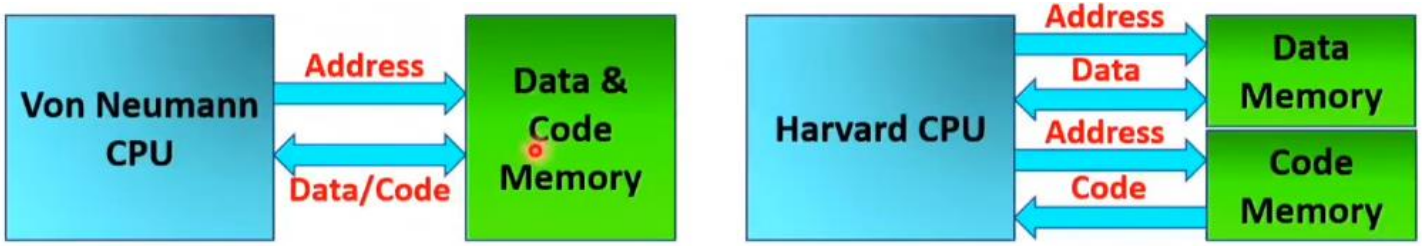


Harvard

Data and instructions are stored into separate memories.



Computer Architecture

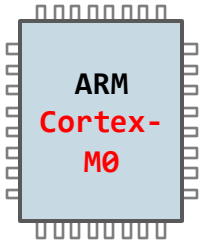


Parameters	Von Neumann	Harvard
Memory	❖ Data and Program {Code} are stored in same memory	❖ Data and Program {Code} are stored in different memory
Memory Type	❖ It has only RAM for Data & Code	❖ It has RAM for Data and ROM for Code
Buses	❖ Common bus for Address & Data/Code	❖ Separate Bus Address & Data/Code
Program Execution	❖ Code is executed serially and takes more cycles	❖ Code is executed in parallel with data so it takes less cycles.
Data/Code Transfer	❖ Data or Code in one cycle	❖ Data and Code in One cycle
Control Signals	❖ Less	❖ More
Space	❖ It needs less Space	❖ It needs more space

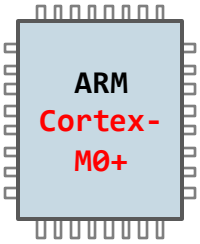
ARM Cortex-M Series Family

Von-Neumann

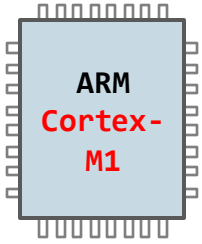
Instructions and data are stored in the same memory.



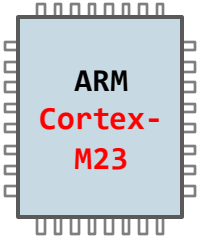
ARMv
6-M



ARMv
6-M



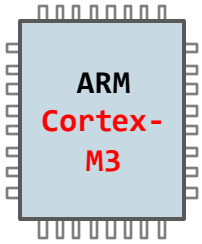
ARMv
6-M



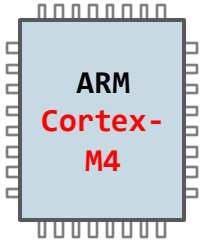
ARMv
8-M

Harvard

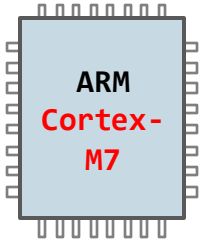
Data and instructions are stored into separate memories.



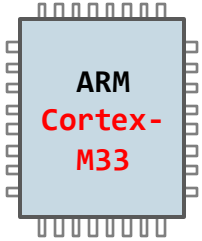
ARMv
7-M



ARMv7
E-M



ARMv7
E-M



ARMv
8-M

Levels of Program Code

C Program

```
int main(void){
  int i;
  int total = 0;
  for (i = 0; i < 10;
i++) {
    total += i;
  }
  while(1); // Dead loop
}
```

Compile

Assembly Program

```
        MOVS r1, #0
        MOVS r0, #0
        B    check
loop    ADD  r1, r1, r0
        ADDS r0, r0, #1
check   CMP  r0, #10
        BLT  loop
self    B    self
```

Assemble

Machine Program

```
0010000100000000
0010000000000000
1110000000000001
0100010000000001
0001110001000000
0010100000001010
1101110011111011
1011111100000000
1110011111111110
```

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

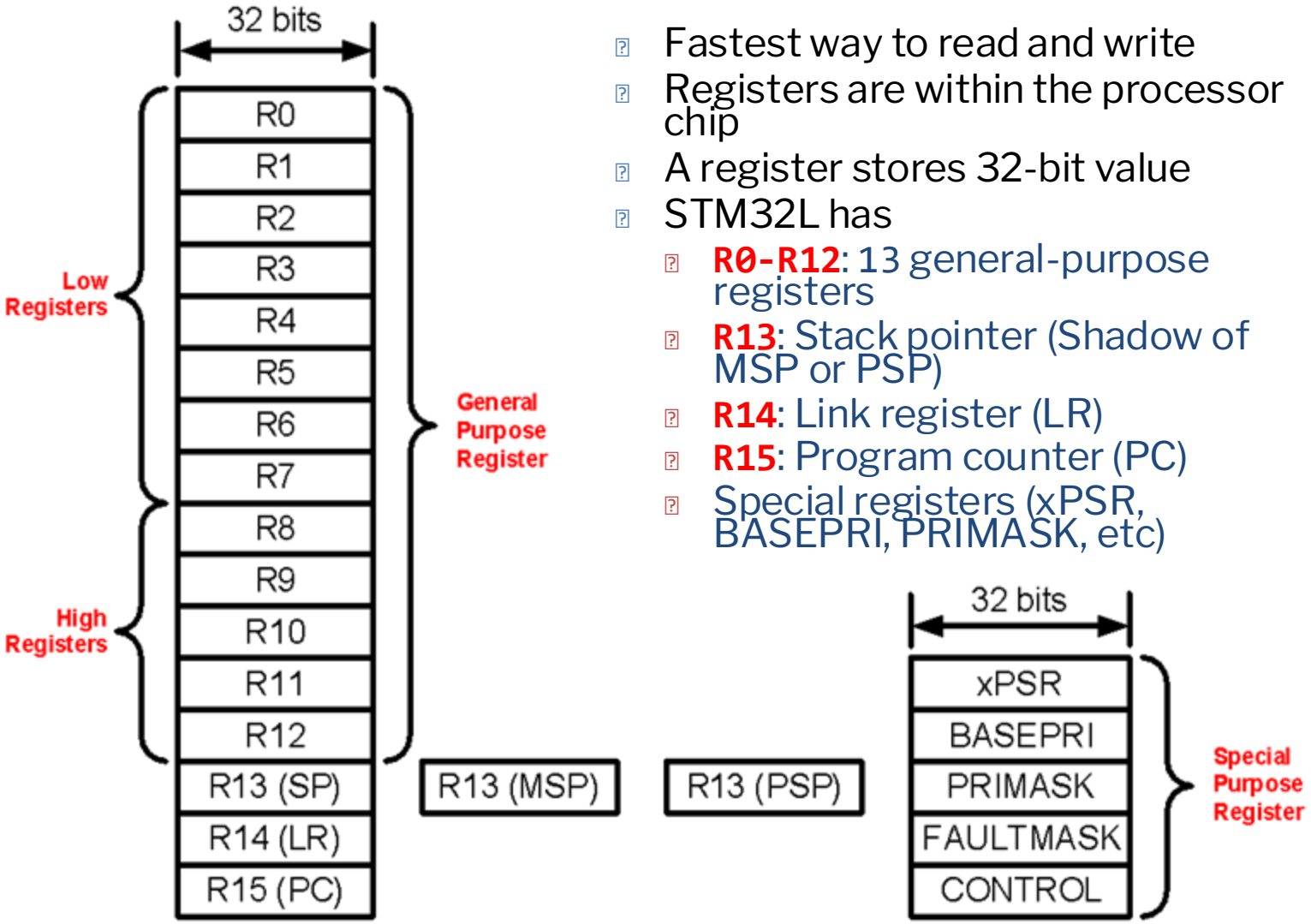
Memory

- linear addressing
- 2 descending stacks
  - main stack
  - process stack – used by OS
- all operations based on registers
- must move data between registers & memory

Registers

- 16 * 32 bit registers
- *working* registers
  - R0-R7 – *low* registers
  - R8-R12 – *high* registers
- R13 – stack pointer (SP)
- R14 - link register (LR)
- R15 – program counter (PC)
- PSR – program status register
  - bits for negative (N), zero (Z), carry (C), overflow (V)

Registers–continued



- Fastest way to read and write
- Registers are within the processor chip
- A register stores 32-bit value
- STM32L has
  - **R0-R12**: 13 general-purpose registers
  - **R13**: Stack pointer (Shadow of MSP or PSP)
  - **R14**: Link register (LR)
  - **R15**: Program counter (PC)
  - Special registers (xPSR, BASEPRI, PRIMASK, etc)

Registers–continued

Stack pointer (SP) r13:

- Holds the memory address of the top of the stack.
- Cortex-M processors provide two different stacks: the main stack and the process stack.
- There are two stack pointers: the main stack pointer (MSP) and the process stack pointer (PSP) .
- The processor uses PSP when executing regular user programs and uses MSP when serving interrupts or privileged accesses.
- The stack pointer (SP) is a shadow register of either MSP or PSP, depending on the processor' s mode setting. When a processor starts, it assigns MSP to SP initially.

Link register (LR) r14:

- Holds the memory address of the instruction that needs to run immediately after a subroutine completes. It is the next instruction after the instruction that calls a subroutine.
- During the execution of an interrupt service routine, LR holds a special value to indicate whether MSP or PSP is used.

Registers–continued

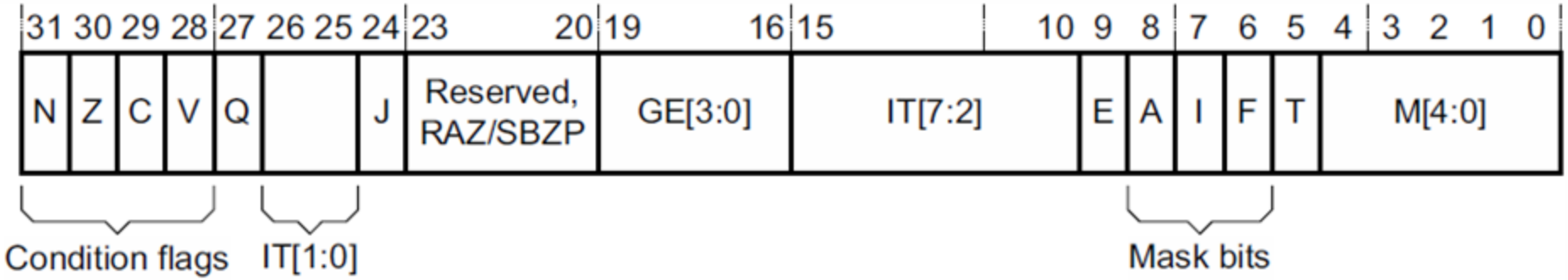
Program counter (PC) r15:
Holds the memory address (location in memory) of the next instruction(s) that the processor fetches from the instruction memory .

Program status register (xPSR):

- Records status bit flags of the application program, interrupt, and processor execution. Example flags include negative, zero, carry,and overflow.

Base priority mask register (BASEPRI)
Priority mask register (PRIMASK)
Fault mask register (FAUL TMASK)

Program Status Registers



- Condition code flags
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed
- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state
- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- Mode bits
  - Specify the processor mode

ARM Documenations

ARM versions:
https://developer.arm.com/documentation/dui0472/m/Compiler-Command-line-Options/--cpu-name

ARM Jazelle State:
https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/The-System-Level-Programmers--Model/Jazelle-direct-bytecode-execution/Jazelle-state?lang=en

Data formats

- *word*
  - 32 bits
  - `long`
- *half word*
  - 16 bits
  - `int`
- byte
  - 8 bits
  - `char`

Programming approach

- in high level programming:
  - identify variables & types
  - compiler maps these to memory & data representation
  - use of memory v registers invisible
- in low level programming
  - must make explicit use of memory & registers
  - must choose data representation
- registers much faster than memory
- try to map variables to registers
- what if more variables than registers…?

Instruction format

A machine instruction consists of:

- a binary operation code (opcode) denoting a specific operation to be carried out
- zero or more operands specifying the inputs of the operation

In an assembly program, each binary opcode is replaced by its symbolic abbreviation, called instruction *mnemonic*. Using human-readable mnemonics instead of binary opcode makes developing an assembly program simpler and more convenient.

```
label       mnemonic operand1, operand2, operand3    ; comments
```

Instruction format

instruction Rd, *Rn*, *operand*$_2$

== *Rd = Rn operation operand*$_2$

- *instruction* is a mnemonic
  - meaningful short form
  - reminds us what instruction does
- *Rd* == destination register - `R0..R15`
- *Rn* == operand register - may be optional
- *operand*$_2$ == *register* or *constant*
- *constant* == #*number*

MOV: move

`MOV` *Rd*, *operand2*

Rd = *operand2*

- don't update condition code flags

`MOVS` *Rd*, *operand2*

- as `MOV` but update condition code flags

`MOV` *Rd*, *#number*

Rd = *number*

NB *number* <= 255 - more later

MVN: move with logical NOT

`MVN` *Rd*, *operand2*

Rd = *operand2*

- don't update condition code flags
- performs a bitwise logical NOT operation on the operand2, and places the result into Rd.

`MVN` *Rd*, *#number*

Rd = !*number*

Label

- Any instruction can be associated with a label

- Example:
- `start: ADD r0,r1,r2 ; a = b+c`

  `next:  SUB r1,r1,#1 ; b--`
- In fact, every instruction has a label regardless if the programmer explicitly names it
  - The label is the address of the instruction
  - A label is a pointer to the instruction in memory

Label

- Therefore, the text label doesn‟t exist in binary code
- relative to program counter (PC)
- turned into PC + offset in machine code
- may start with _ such as _global

System calls

- `SWI 0` == software interrupt to call function
- parameters in R0-R2
- system function number in R7
- e.g. `exit` is `1`

- NB shell variable `$?` is R0 so:

  `echo $?` == display R0

Layout

- usual to precede instruction with a *tab*
- comment is: @ *text // text*
- large comment: /*.....*/

Program skeleton

```
.global _start
_start:
@ program code here

_
     MOV R0, #65 @ arbitrary value
     MOV R7, #1
     SWI 0
```

- `.global _start`
  - assembler directive
  - entry point `_start` is visible externally

Running assembler programs

- *file*`.s` == assembly language source

`$ as -o` *file*`.o` *file*`.s`

- assemble *file*`.s` to *file*`.o`

`$ ld -o` *file file*`.o`

- create executable *file* from *file*`.o`

`$ ./file`

- run executable in *file* from `_start`

`$ echo $?`

- display R0

Example: exit

- suppose `exit.s` is basic start

$ program

```
$ as -o exit.o exit.s

  ld -o exit exit.o
$ ./exit
$ echo $?
65
```

ADD: addition

`ADD` *Rd*, *Rn*, *operand*$_2$

Rd = *Rn*+*operand*$_2$

`ADD` *Rd*, *operand*$_2$

Rd =*Rd*+*operand*$_2$

`ADDC`

- add with carry
- like `ADD +` carry flag

`ADDS/ADDCS`

- like `ADD/ADDC` but set condition code flags

SUB: subtraction

`SUB` *Rd*, *Rn*, *operand*$_2$

Rd = *Rn* − *operand*$_2$

`SUB` *Rd*, *operand*$_2$

Rd = *Rd* − *operand*$_2$

`SBC`

- subtract with carry
- like `SUB -1` if carry flag not set

`SUBS/SBCS`

- like `SUB/SUBC` but set condition code flags

RSB: reverse subtraction

`RSB` *Rd*, *Rn*, *operand*$_2$

Rd = *operand*$_2$ − ***Rn***

`RSB` *Rd*, *operand*$_2$

Rd = *operand*$_2$ - *Rd*

Multiplication

- In ARM, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
  - Syntax of Multiplication (signed):
    - `MUL` register1, register2, register3

- Built in multiplication `MUL`
  - {*Rd*, } *Rn*, *operand*$_2$? *Rd* = *Rn***operand*$_2$
  - NB *Rd* cannot be an *operand*

Multiplication

- Example:
  - in C: `a = b * c;`
  - in MIPS:
    - let b be r2; let c be r3; and let a be r0 and r1 (since it may be up to 64 bits)

```
MUL r0, r2, r3      ; b*c only 32 bits
stored
```

Note: Often, we only care about the lower half of the product.

```
SMULL r0,r1,r2,r3 ; 64 bits in r0:r1
```

Multiply and Divide

- There are 2 classes of multiply - producing 32-bit and 64-bit results
- 32-bit versions on an ARM7TDMI will execute in 2 - 5 cycles

```
MUL r0, r1, r2          ; r0 = r1 * r2
MLA r0, r1, r2, r3      ; r0 = (r1 * r2) + r3
```

- 64-bit multiply instructions offer both signed and unsigned versions
  - For these instruction there are 2 destination registers

```
[U|S]MULL r4, r5, r2, r3      ; r5:r4 = r2 * r3
[U|S]MLAL r4, r5, r2, r3      ; r5:r4 = (r2 * r3) + r5:r4
```

- Most ARM cores do not offer integer divide instructions
  - Division operations will be performed by C library routines or inline shifts

Instruction Summary So far

- In ARM Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- Instructions so far:

  `ADD, SUB, RSB, MUL, MULA, [U|S]MULL, [U|S]MLAL`
- Registers:

  Places for general variables: `r0-r12`

Expressions

- for:

$var_1 = var_2 \; op \; var_3$

- with:

var_1 in R_1

var_2 in R_2

var_3 in R_3

 op R_1, R_2, R_3

- e.g.

```
x = y*z;
```

```
@ x == R1
@ y == R2
@ z == R3
MUL R1,R2,R3
```

Expressions

- for:

$var_1 = var_2\, op_1\, var_3\, op_2\, var_4$

- if op_1 and op_2 have same precedence
- or op_1 has higher precedence than op_2

$op_1\, R_1, R_2, R_3$

$op_2\, R_1, , R_4$

e.g.

```
x = y*z-a;
...
@ a == R4
MUL R1,R2,R3
SUB R1,R4
```

Expressions

- for:

$var_1 = var_2\, op_1\, var_3\, op_2\, var_4$

- if op_2 has higher precedence than op_1
- must evaluate *op2* expression in new register

$op_2\ R_i, R_3, R_4$

$op_1\, R_1, , R_2, R_i$

e.g.

```
x = y+z*a;
```

```
MUL R5,R3,R4
```

```
ADD R1,R2,R5
```

e.g.

```
x = y*(z+a)
```

```
ADD R5,R3,R4
```

```
MUL R1,R2,R5
```

CMP: compare

`CMP` *Rd*, *operand*$_2$

- subtract *operand*$_2$ from *Rd* BUT …
- … do not modify *Rd*
  - otherwise same as `SUBS`
- update Z, N, C, V flags

`CMN` *Rd*, *operand*$_2$

- add *operand*$_2$ to *Rd* BUT …
- … do not modify *Rd*
  - otherwise same as `ADDS`
- update Z, N, C, V flags

Flags

- N – 1 if result <0; 0 otherwise
- Z – 1 if result =0; 0 otherwise
- C – 1 if result led to carry; 0 otherwise
  - i.e. $X+Y > 2^{32}$
  - i.e. X-Y >= 0

Flags

- V – 1 if result led to overflow; 0 otherwise
  - (-) == negative
  - (+) == positive
  - i.e. (-)X+(-)Y > 0
  - i.e. (+)X+(+)Y < 0
  - i.e. (-)X-(+)Y > 0
  - i.e. (+X)-(-Y) < 0

B: branch

`B` *label*

- branch to *label*
- i.e. reset PC to address for *label*

`B`*cond label*

- branch on *condition* to *label*

C Decisions: `if` Statements

- `if` statements in C
  - `if (`*condition*`)` *clause*
  - `if (`*condition*`)` *clause1* `else` *clause2*
- Rearrange 2nd `if` into following:

```
if (condition) goto L1;
    clause2;
    goto L2;
L1: clause1;
L2:
```

  - Not as elegant as if-else, but same meaning

ARM `goto` Instruction Conversion

- The simplest control instruction is equivalent to a C `goto` statement
- `goto label` (in C) is the same as:
- `B label` (in ARM)
- `B` is shorthand for "branch". This is called an unconditional branch meaning that the branch is done regardless of any conditions.
- There are also conditional branches

Condition

suffix	flags
`EQ` == equal	Z=1
`NE` == not equal	Z=0
`CS/HS` == carry set/ higher or same - unsigned	C=1
`CC/LO` == carry clear/lower - unsigned	C=0
`MI` == negative	N=1
`PL` == positive or 0	N=0
`VS` == overflow	V=1
`VC` == no overflow	V=0

Condition

suffix	flags
`HI` == higher - unsigned	C=1 & Z=0
`LS` == lower or same - unsigned	C=0 or Z=1
`GE` == greater than or equal - signed	N=V
`LT` == less than - signed	N!=V
`GT` == greater than - signed	Z=0 & N=V
`LE` == less than or equal, signed	Z=1 or N!=V
`AL` == any value	default if not *cond*

Overflow Detection

One way to detect overflow is to check whether the sign bit is consistent with the sign of the inputs when the two inputs are of the same sign – if you added two positive numbers and got a negative number, something is wrong, and vice versa.

A+B			
Sign of A	Sign of B	Sign of Result	Overflow?
+	+	-	Yes
-	-	+	
+	+	+	No
-	-	-	
+	-	-	No
+	-	+	No
-	+	-	No
-	+	+	No

ARM BRANCH Instructions

- ARM also has variants of the branch instruction that only goto the label if a certain condition is TRUE
- Examples:
  - `BEQ label ; BRANCH EQUAL`
  - `BNE label ; BRANCH NOT EQUAL`
  - `BLE label ; BRANCH LESS THAN EQUAL`
  - `BLT label ; BRANCH LESS THAN`
  - `BGE label ; BRANCH GREATER THAN EQUAL`
  - `BGT label ; BRANCH GREATER THAN`
  - Plus more …
- The condition is T/F based upon the fields in the Program Status Register

Example: multiply by adding

```
    int x;
   int y;
    int m;
x = 3;
y = 10;
m = 0;
while(y!=0)
{     m = m+x;
y = y-1;
}
```

```
.global _start
_start:
  MOV R1, #0x03
  MOV R2, #0x0a
  MOV R3, #0x00
_loop:
  CMP R2,#0x00
  BEQ _exit
  ADD R3, R1
  SUB R2, #0x01
  B  _loop
```

Example: multiply by adding

```
   #m=x*y
   int x;
   int y;
   int m;
x = 8;
y = 10;
m = 0;
while(y!=0)
{   m = m+x;
    y = y-1;
}
```

```
_exit:
  MOV R0, R3
  MOV R7, #1
  SWI 0
...
$ echo $?
12
```

Data directives & constants

`.data`

- start of sequence of data directives
- usually after instuctions program

`.equ` *label*`,` *value*

- define constant
- associate *label* with *value*
- use #*label* as $operand_2$ to get *value*

Example: multiply by adding

- m == x*y

```
int x;  int y;
int m;
x = 8;
y = 10;
m = 0;
while(y!=0)
{     m = m+x;
    y = y-1;
}
```

```
.global _start
_start:
  MOV R1, #X
  MOV R2, #Y
  MOV R3, #0x00
_loop:
  CMP R2,#0x00
  BEQ _exit
  ADD R3, R1
  SUB R2, #0x01
  B  _loop
```

Example: multiply by adding

```
m=x*y
 int x;
int y;
int m;
x = 8;
y = 10;
m = 0;
while(y!=0)
{   m = m+x;
    y = y-1;
}
```

```
_exit:
  MOV R0, R3
  MOV R7, #1
  SWI 0
.data
.equ X, 3
.equ Y, 4
...
$ echo $?
12
```

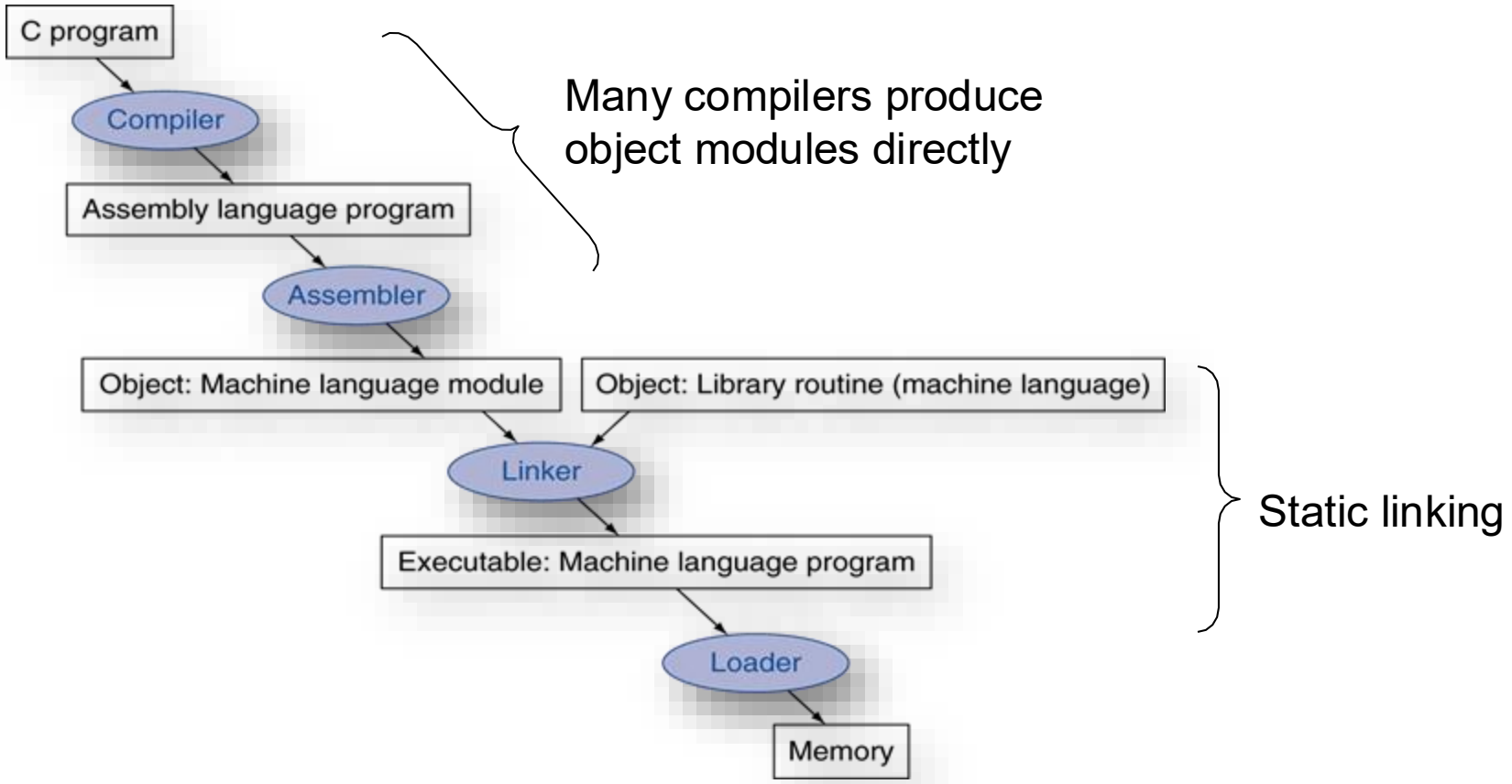
Tracing and debugging

- `gdb` - GNU debugger
- assemble with `-g` flag
- run program from `gdb`

```
$ as -g -o file.o file.s
$ ld -o file file.o
```

```
$ ./file
gdb file
...
```

Translation and Startup



UNIX: C source files are named x.c, assembly files are x.s, object files are named x.o, statically linked library routines are x.a, dynamically linked library routes are x.so, and executable fi les by default are called a.out.
MS-DOS uses the .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect.

Tracing and debuggging

- `q(uit)` - return to shell
- `l(ist)` - show program - may need to press return
- `r(un)` - run program
- `b(reak)` *number* – set breakpoint at line *number*
  - program pauses at breakpoint
  - can set multiple breakpoints
  - can also set break points at labels & addresses

Tracing and debuggging

- `i(nfo) r` – show registers
- `c(ontinue)` - continue execution after breakpoint
- `s(tep)` - execute next instruction
- `i(nfo) b` - show breakpoints
- `d(elete)` *number* – remove breakpoint *number*

Tracing and debugging

- to trace
  - set breakpoint at start of program
  - step & show registers
- to debug
  - set breakpoints at salient points in program
  - e.g. at start or ends of loops
  - continue/step & show registers

Tracing and debugging

- e.g. suppose mult program is in `mult.s`

```
$ as -g -o mult.o mult.s
$ ld -o mult mult.s
$ gdb mult
(gdb) l
1       .global _start
2
3  _start:
4           MOV R1, #X
5           MOV R2, #Y
6           MOV R3, #0x00
7  _loop:
8           CMP R2,#0x00
9           BEQ _exit
10          ADD R3, R1
(gdb)
11          SUB R2, #0x01
12          B   _loop
13 _exit:
14          MOV R0, R3
15          MOV R7, #1
16          SWI 0
17
18 .data
19 .equ X, 3
20 .equ Y, 4
(gdb)
```

Tracing & debugging

```
(gdb) b _start
Breakpoint 1 at 0x805b:
file mult.s, line 5
(gdb) r
...
5       MOV R2, #Y
(gdb) i r
r0      0x0     0
r1      0x3     3
r2      0x0     0
r3      0x0     0
(gdb) s
6         MOV R, #0
```

```
(gdb) i r
r0      0x0     0
r1      0x3     3
r2      0x0     4
r3      0x0     0
...
(gdb) b 12
Breakpoint 2 at 0x8070:
file mult.s, line 12
(gdb) c
...
12      B _loop
```

Tracing & debugging

```
(gdb) i r
r0      0x0    0
r1      0x3    3
r2      0x0    3
r3      0x0    3
...
(gdb) c
...
12        B _loop
(gdb) i r
r0      0x0    0
r1      0x3    3
r2      0x0    2
r3      0x0    6
...
```

```
(gdb) c
...
12        B _loop
(gdb) i r
r0      0x0    0
r1      0x3    3
r2      0x0    1
r3      0x0    9
...
(gdb) c
...
12        B _ loop
```

Tracing & debugging

```
(gdb) i r
r0      0x0    0
r1      0x3    3
r2      0x0    0
r3      0x0    12
...
(gdb) c
...
[Inferior 1 (process 2614)
exited with code 014]
(gdb) q
$
```

NOP: no operation

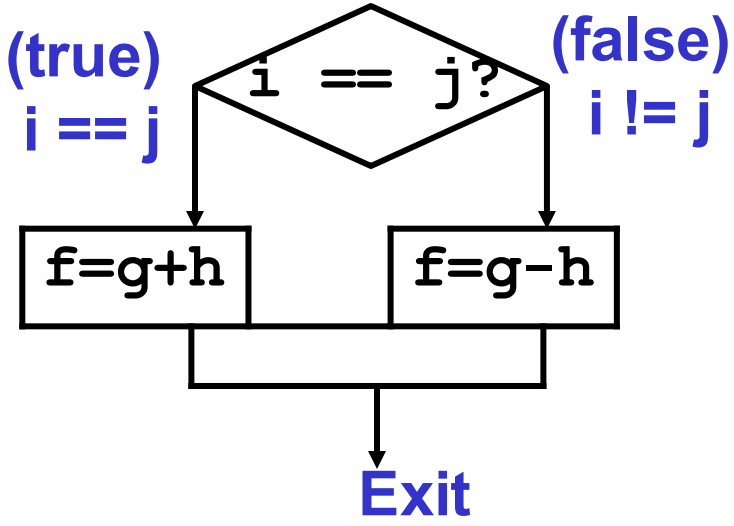
```
NOP
```

- do nothing
- use for padding/layout
- no cost

Compiling C `if` into ARM

- Compile by hand

```
if (i == j) f=g+h;
else
f=g-h;
```



- Use this mapping:

`f: r0, g: r1, h: r2, i: r3, j: r4`

Comparison Instructions

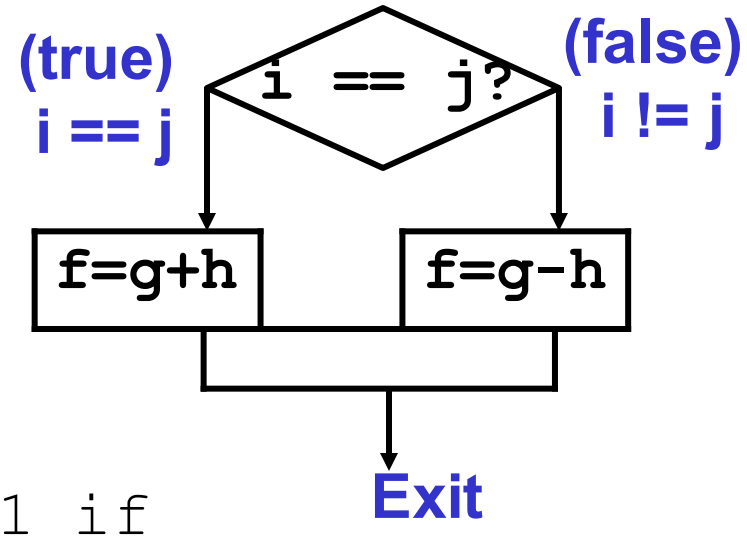
- In order to perform branch on the "==" operation we need a new instruction
- `CMP` – Compare: subtracts a register or an immediate value from a register value and updates condition codes
- Examples:
  - `CMP r3, #0 ; set Z flag if r3 == 0`
  - `CMP r3, r4 ; set Z flag if r3 == r4`

All flags are set as result of this operation, not just Z.

Compiling C `if` into ARM

Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```



- Final compiled ARM code:

```
      CMP   r3, r4      ; Z = 1 if
      i==j  BEQ True    ; goto True when i==j
      SUB   r0,r1,r2    ; f=g-h(false)
      B     Fin         ; goto Fin
True: ADD   r0,r1,r2    ; f=g+h (true)

Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches) appropriately. Generally not found in C code.

Loops in C/Assembly

- Simple loop in C;

```
do{
   g--;
   i = i + j;}
while (i != h);
```

- Rewrite this as:

```
Loop: g--;
i =   i + j;
if   (i != h)  goto Loop;
```

- Use this mapping:

```
g: r1, h: r2,i: r3, j:r4
```

Loops in C/Assembly

- Final compiled ARM code:

```
Loop    SUB r1,r1,#1     ; g--
        ADD r3,r3,r4     ; i=i+j
        CMP r3,r2        ; cmp i,h
        BNE Loop         ; goto Loop
                         ; if i!=h
```

Inequalities in ARM

- Until now, we‟ve only tested equalities (== and != in C). General programs need to test < and > as well.
- Use `CMP` and `BLE, BLT, BGE, BGT`
- Examples:

```
if (f < 10) goto Loop; =>   CMP r0,#10
                            BLT Loop
if (f >= i) goto Loop; =>   CMP r0,r3
                            BGE Loop
```

Loops in C/Assembly

- There are three types of loops in C:
  - `while`
  - `do... while`
  - `for`
- Each can be rewritten as either of the other two, so the method used in the previous example can be applied to `while` and `for` loops as well.
- Key Concept: Though there are multiple ways of writing a loop in ARM, conditional branch is key to decision making

Example: The C Switch Statement

- Choose among four alternatives depending on whether `k` has the value 0, 1, 2 or 3. Compile this C code:

```
switch (k) {
   case 0: f=i+j; break; /* k=0*/
   case 1: f=g+h; break; /* k=1*/
   case 2: f=g-h; break; /* k=2*/
   case 3: f=i-j; break; /* k=3*/
   }
```

Example: The C Switch Statement

- This is complicated, so simplify.
- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if(k==0) f=i+j;
  else if(k==1) f=g+h;
    else if(k==2) f=g-h;
      else if(k==3) f=i-j;
```

- Use this mapping:

```
f: $r0, g: $r1, h: $r2, i: $r3,
 j: r4, k: $r5
```

Example: The C Switch Statement

```
      CMP  r5,#0          ; compare k, 0
      BNE  L1             ; branch k!=0
      ADD  r0,r3,r4       ; k==0 so f=i+j
      B    Exit           ; end of case so Exit
L1    CMP  r5,#1          ; compare k, -1
      BNE  L2
      ADD  r0,r1,r2       ; k==1 so f=g+h
      B    Exit           ; end of case so Exit
L2    CMP  r5,#2          ; compare k, 2
      BNE  L3             ; branch k!=2
      SUB  r0,r1,r2       ; k==2 so f=g-h
      B    Exit           ; end of case so Exit
L3    CMP  r5,#3          ; compare k, 3
      BNE  Exit           ; branch k!=3
      SUB  r0,r3,r4       ; k==3 so f=i-j
Exit:
```

Predicated Instructions

- All instructions can be executed conditionally.

Simply add {EQ,NE,LT,LE,GT,GE, etc.} to end

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else:
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1
...
```

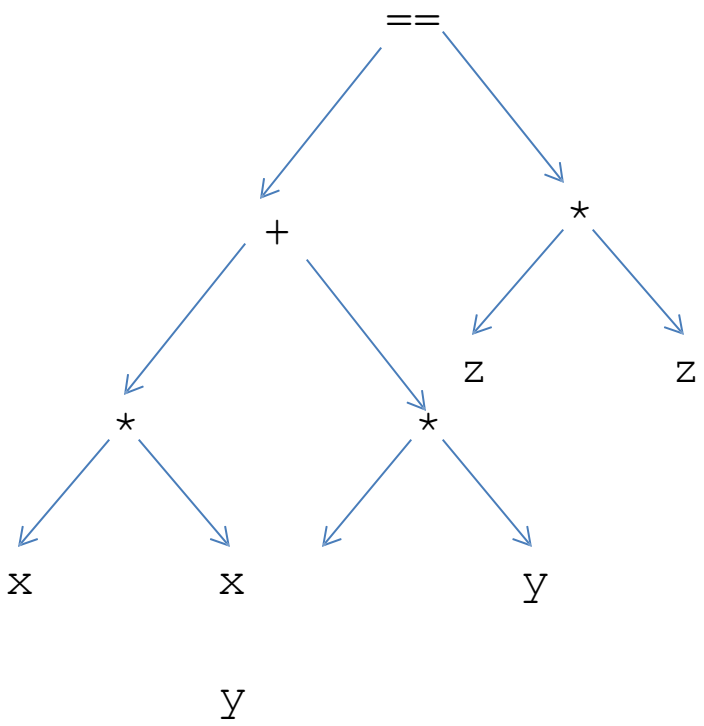
- 3 instructions
- 3 words
- 3 cycles

Expressions

- draw expression tree
- allocate registers to nodes
  - from bottom left
  - if expression in assignment then start with register for destination variable
- accumulate into register for left operand
- can re-use any register whose value is no longer required

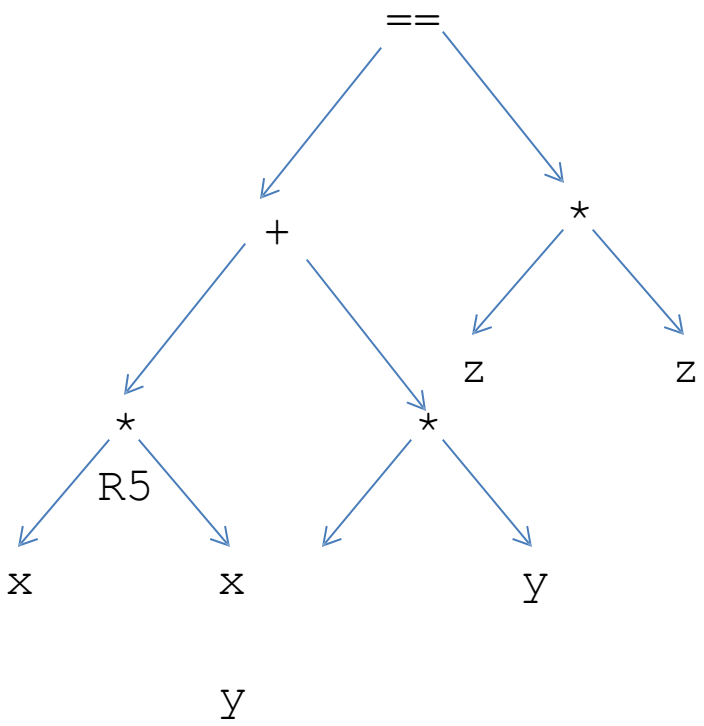
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



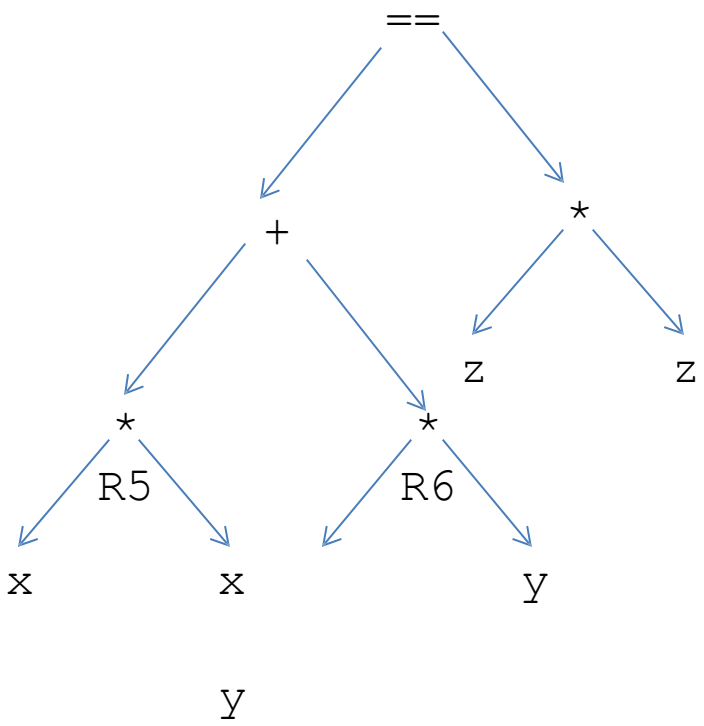
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



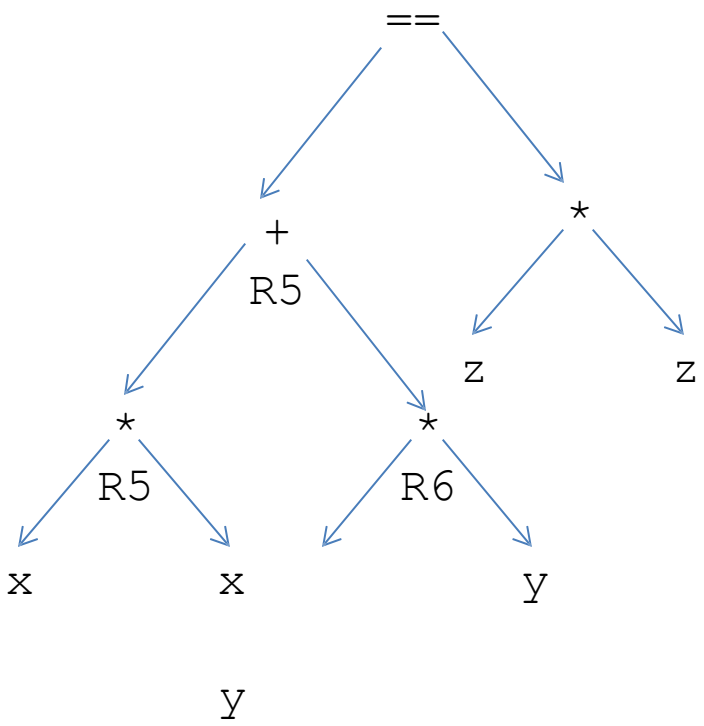
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



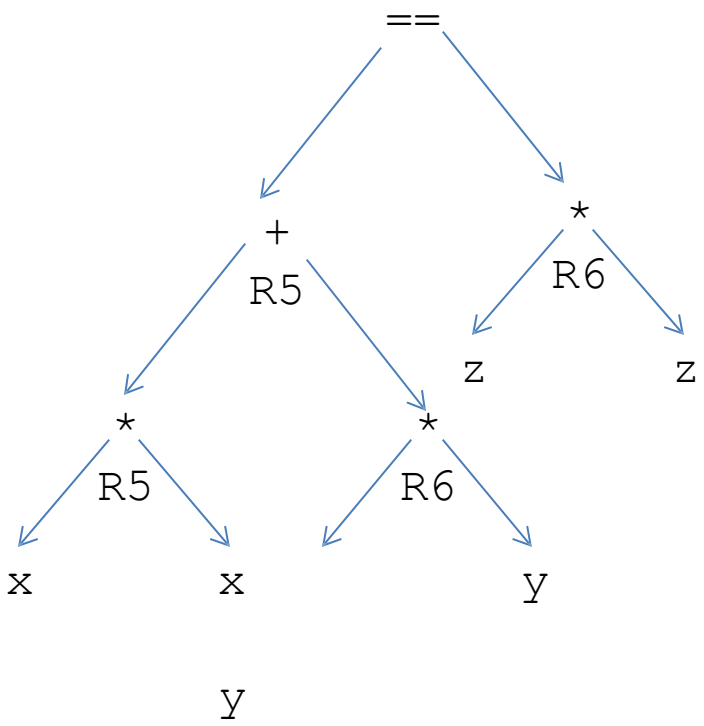
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



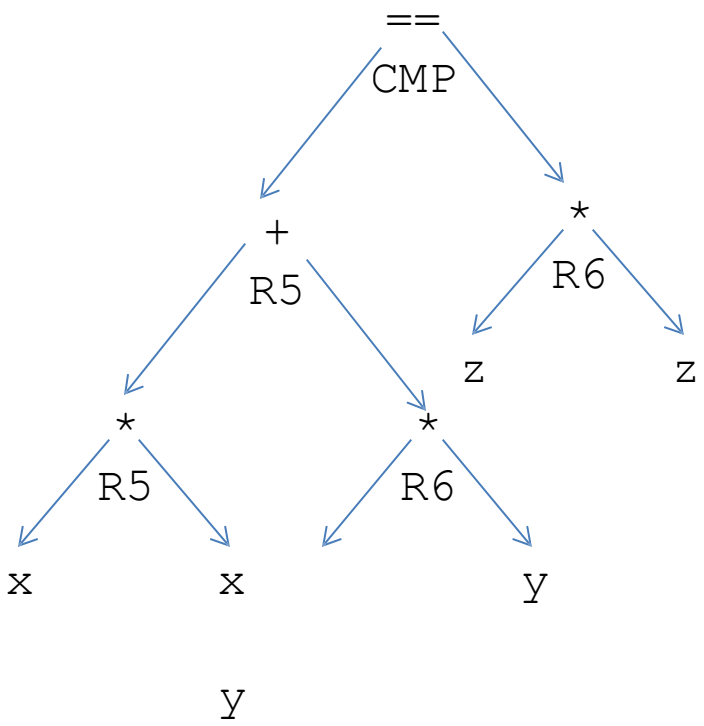
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



Example: Pythagoras

```
int x;
int y;
int z;
int p;
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```

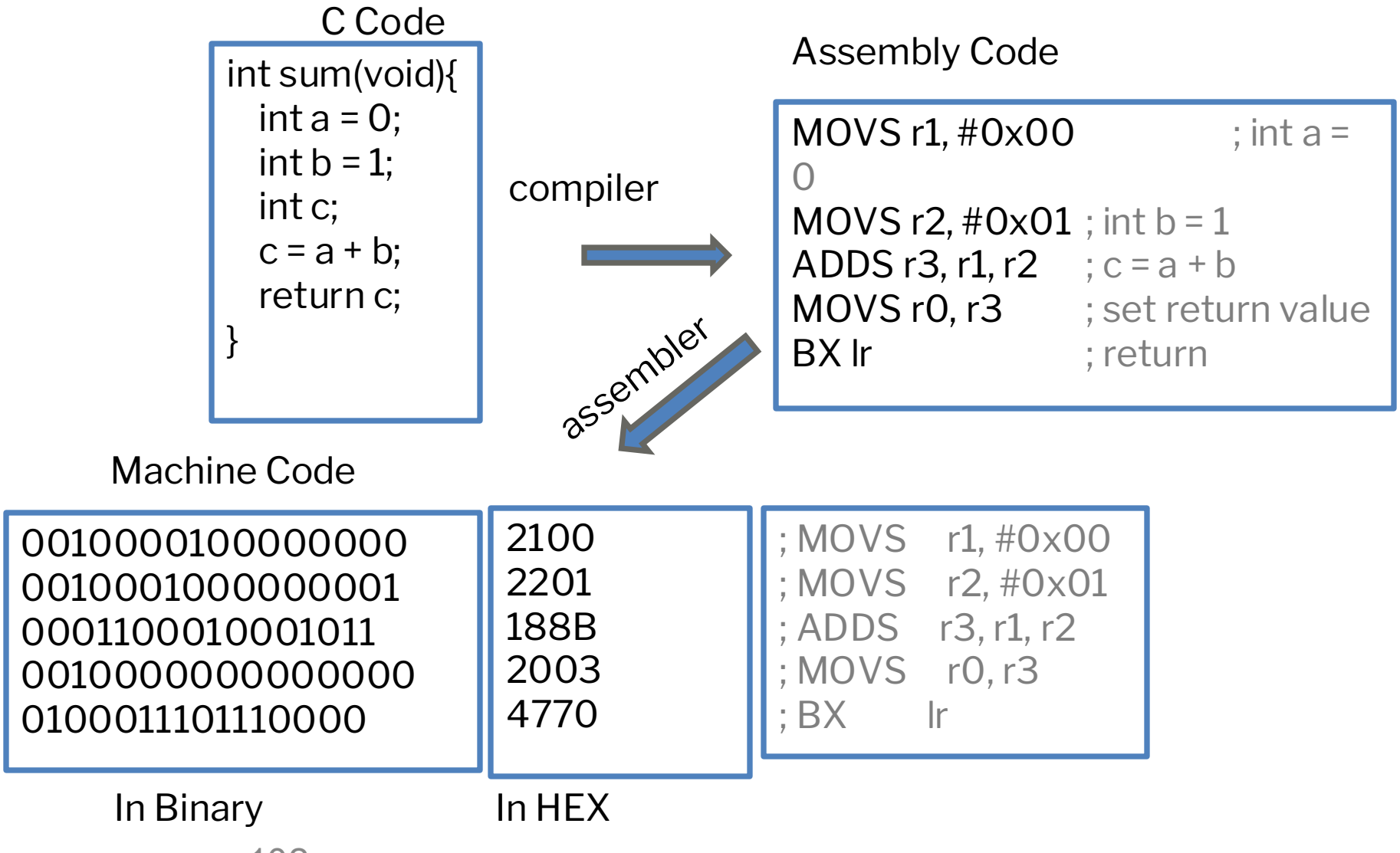
```
.global _start
_start:
        MOV R1, #X
        MOV R2, #Y
        MOV R3, #Z
        MUL R5, R1, R1
        MUL R6, R2, R2
        ADD R5, R6
        MUL R6, R3, R3
        CMP R5, R6
        BEQ _same
        MOV R4, #0
        B _exit
_same:
        MOV R4, #1
```

Example: Pythagoras

```
int x;
int y;
int z;
int p;
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```

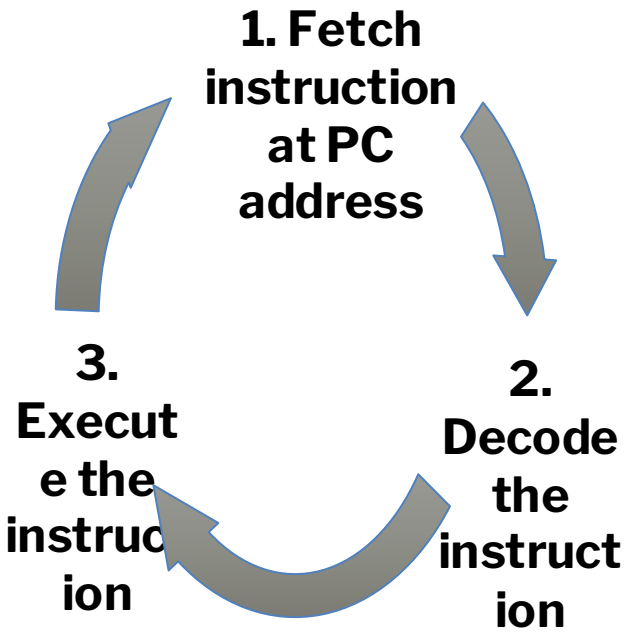
```
_exit:
        MOV R0,  R4
        MOV R7,  #1
        SWI 0
.data
.equ X,  3
.equ Y,  4
.equ Z,  5
```

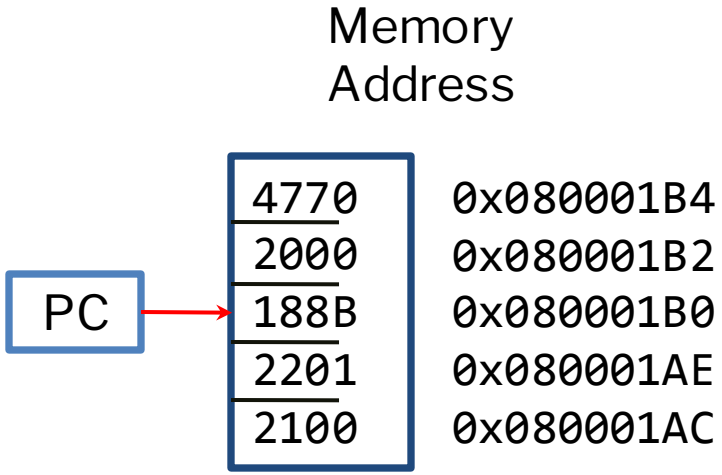
See a Program Runs



Program Execution

Program Counter (PC) is a register that holds the memory address of the next instruction to be fetched from the memory.



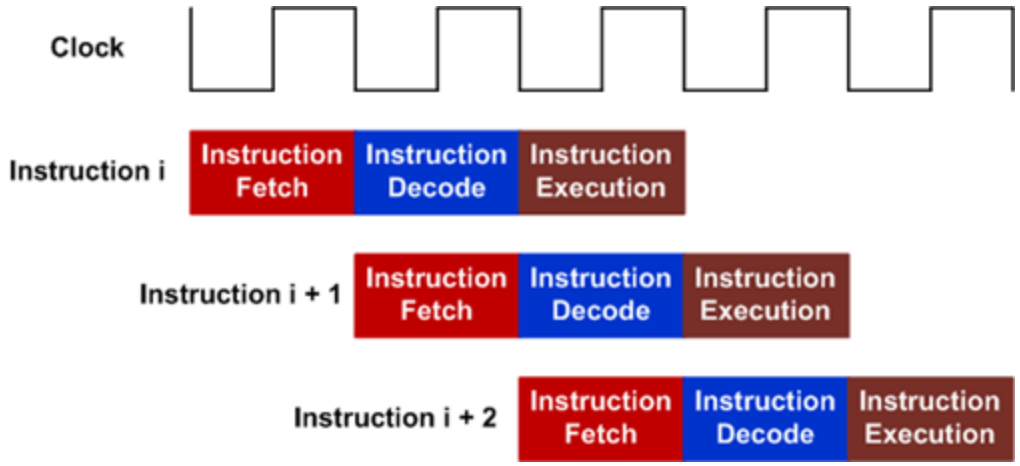


PC = 0x080001B0
Instruction = 188B

Three-state pipeline: Fetch, Decode, Execution

Pipelining allows hardware resources to be fully utilized

One 32-bit instruction or two 16-bit instructions can be fetched.

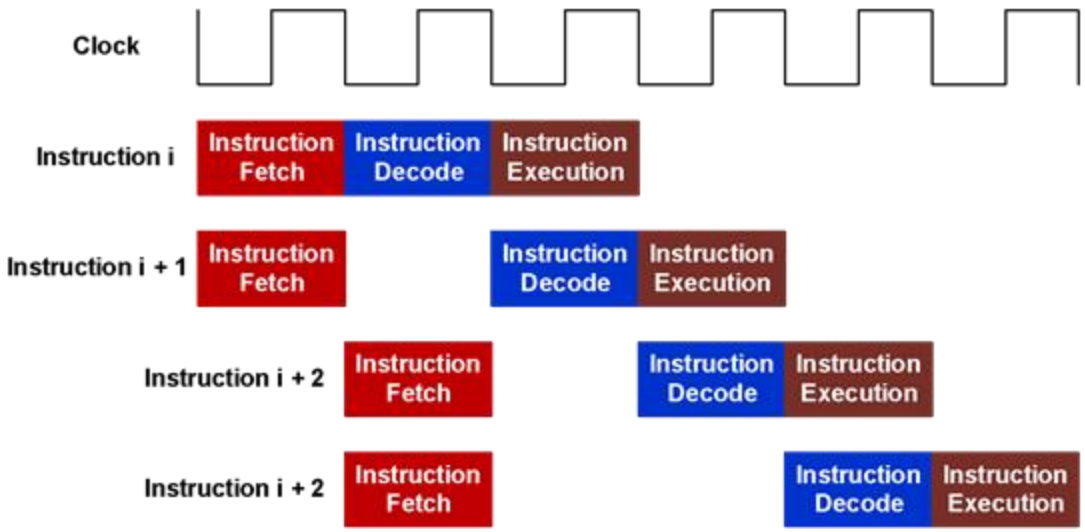


Pipeline of 32-bit instructions

Three-state pipeline: Fetch, Decode, Execution

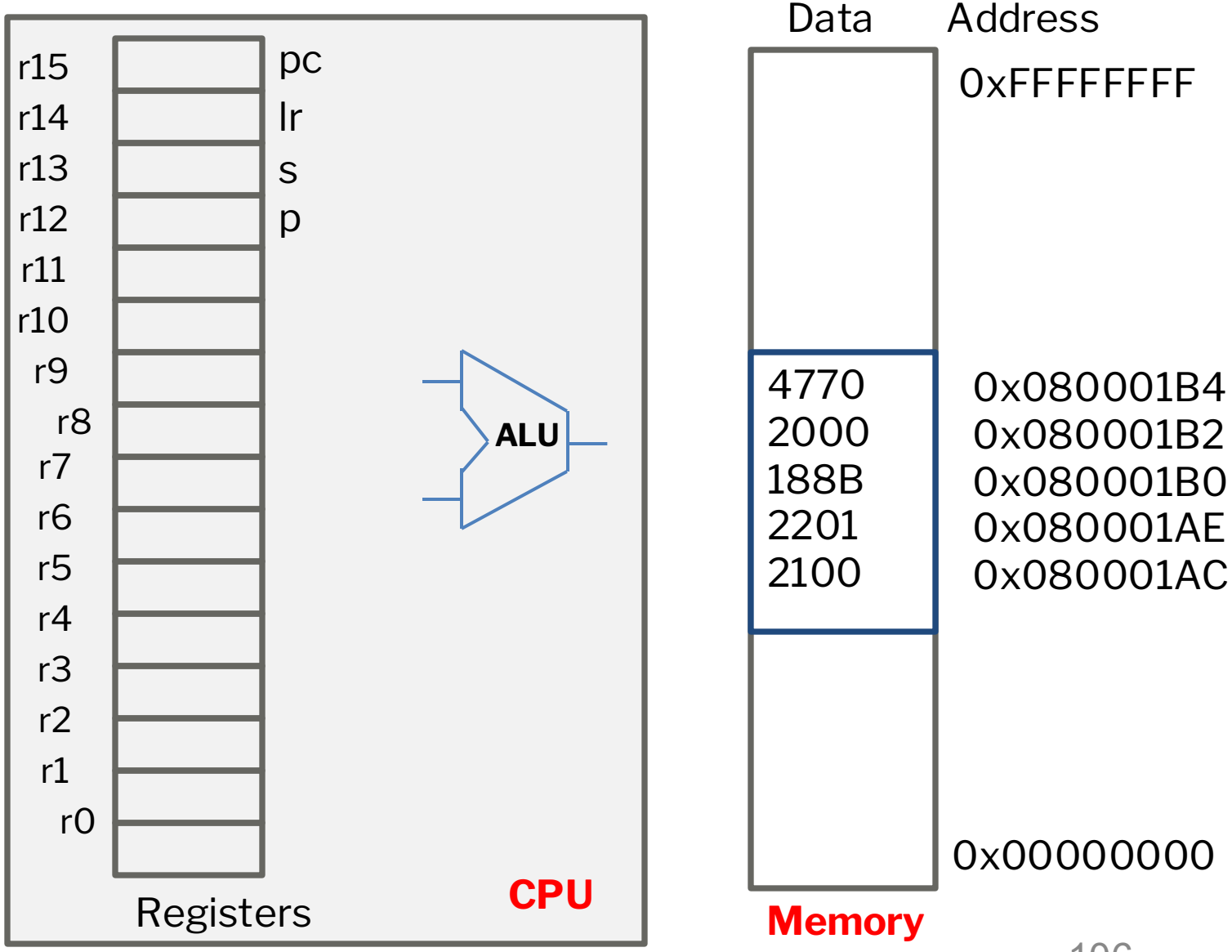
Pipelining allows hardware resources to be fully utilized

One 32-bit instruction or two 16-bit instructions can be fetched.



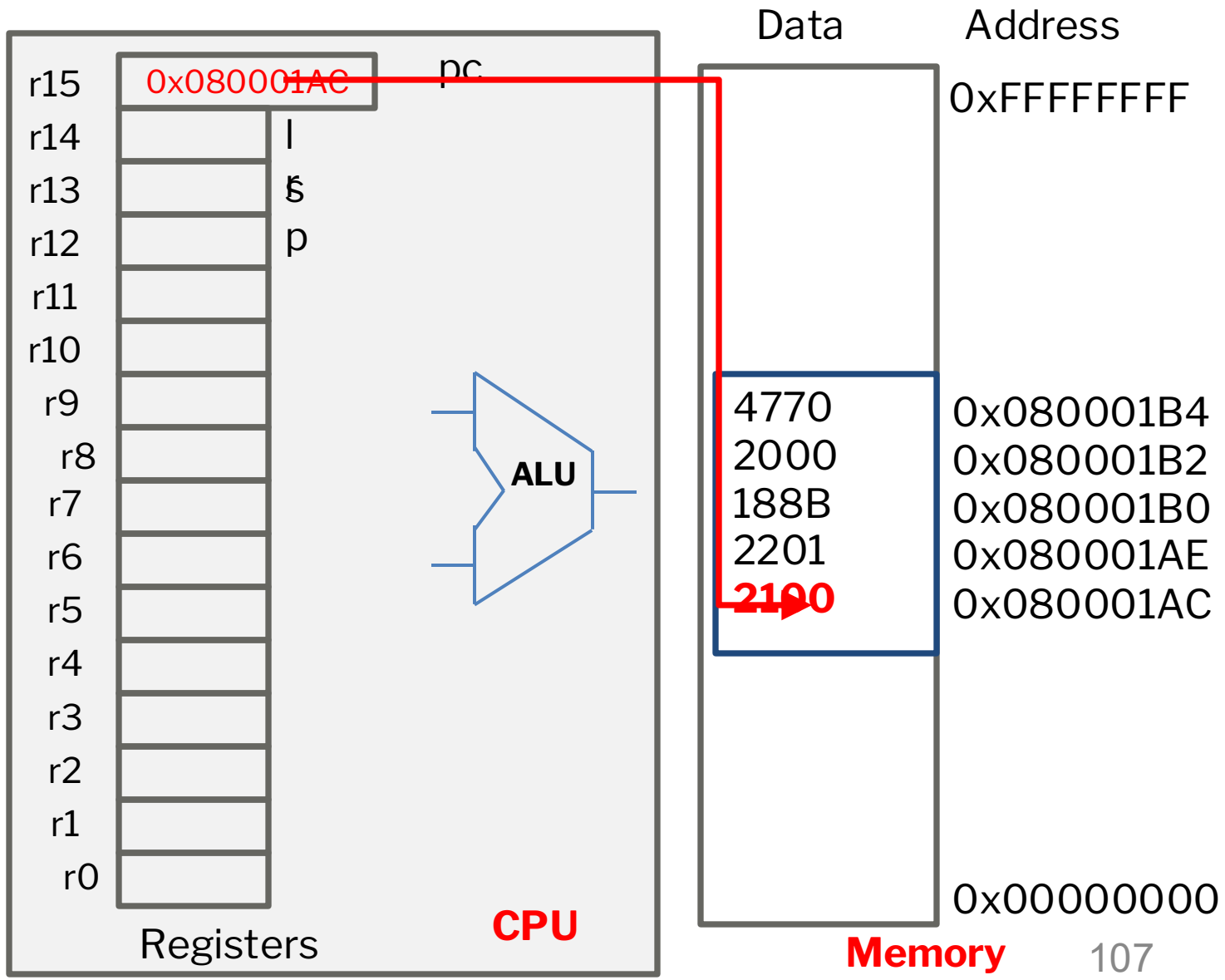
Pipeline of 16-bit instructions

Machine codes are stored in memory

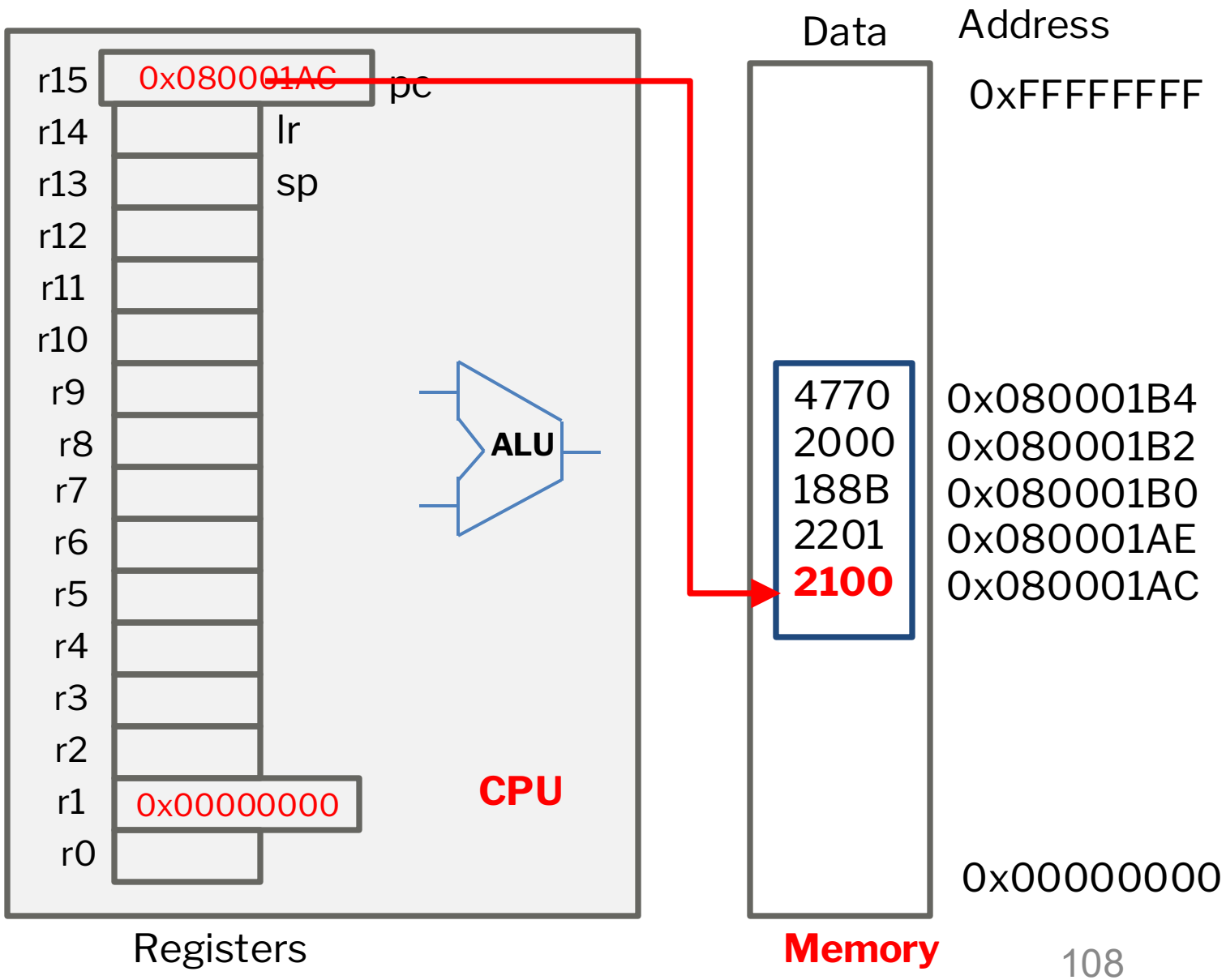


Fetch Instruction: pc = 0x08001AC

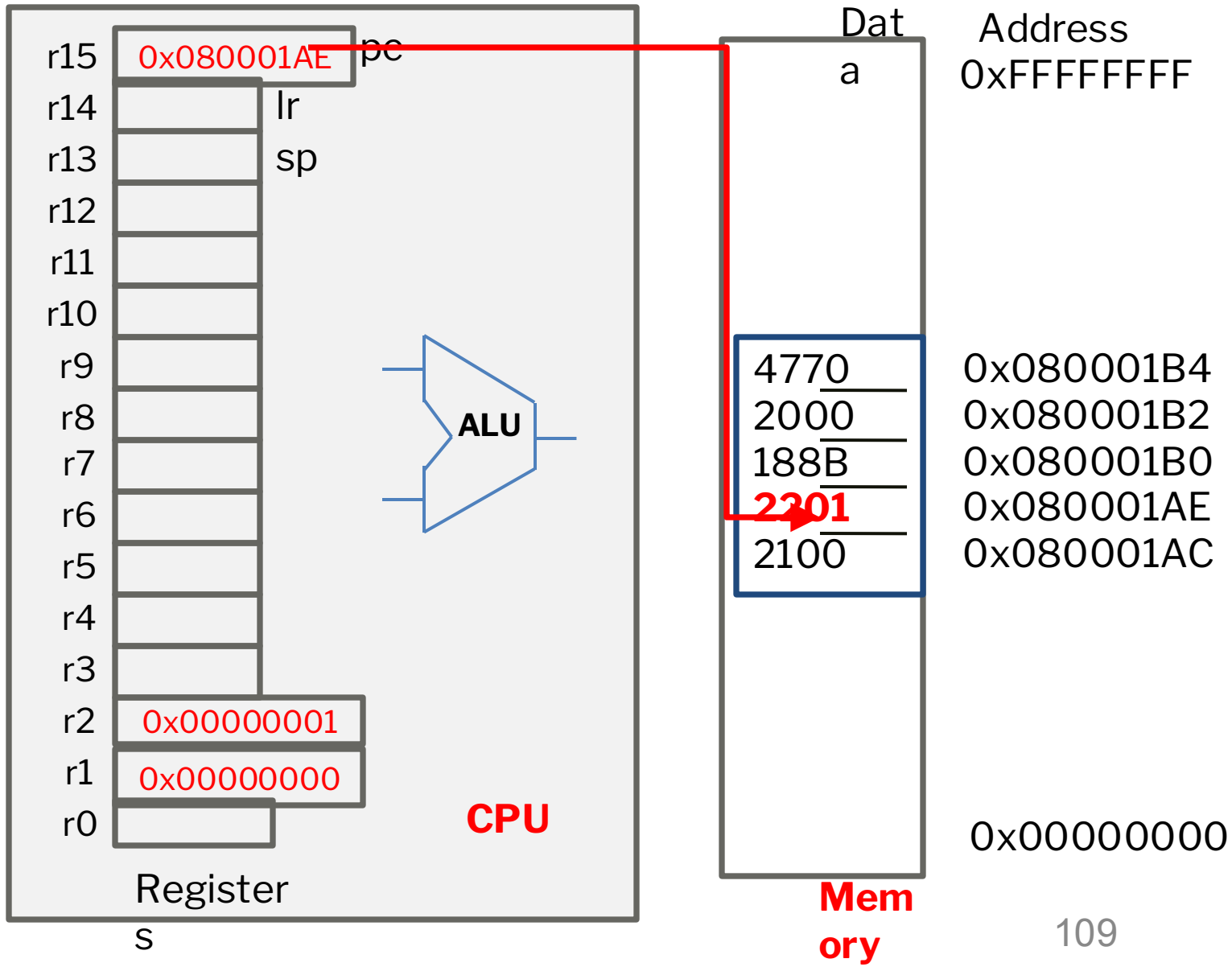
Decode Instruction: 2100 = MOVS r1, #0x00



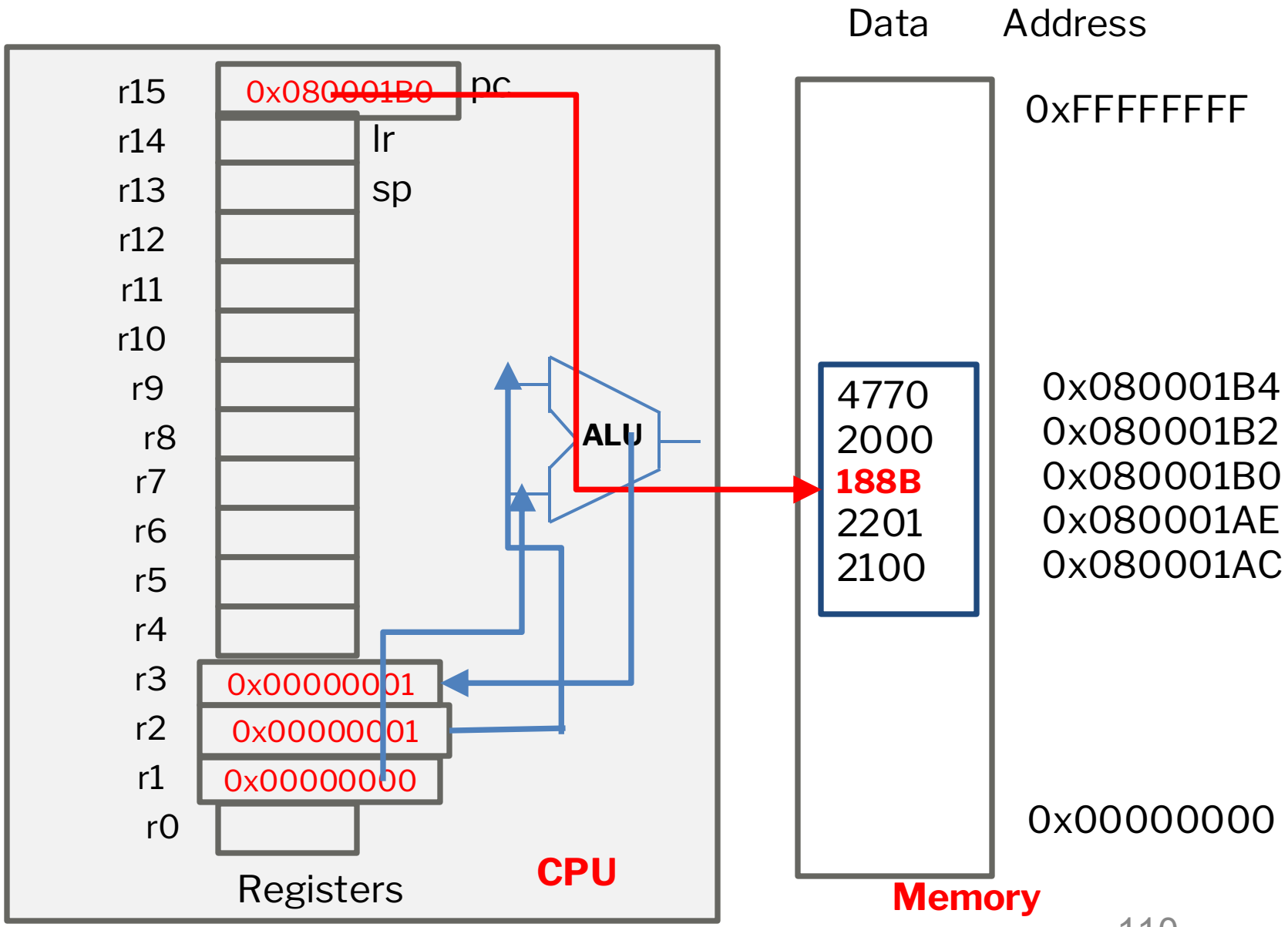
Execute Instruction: MOVS r1, #0x00



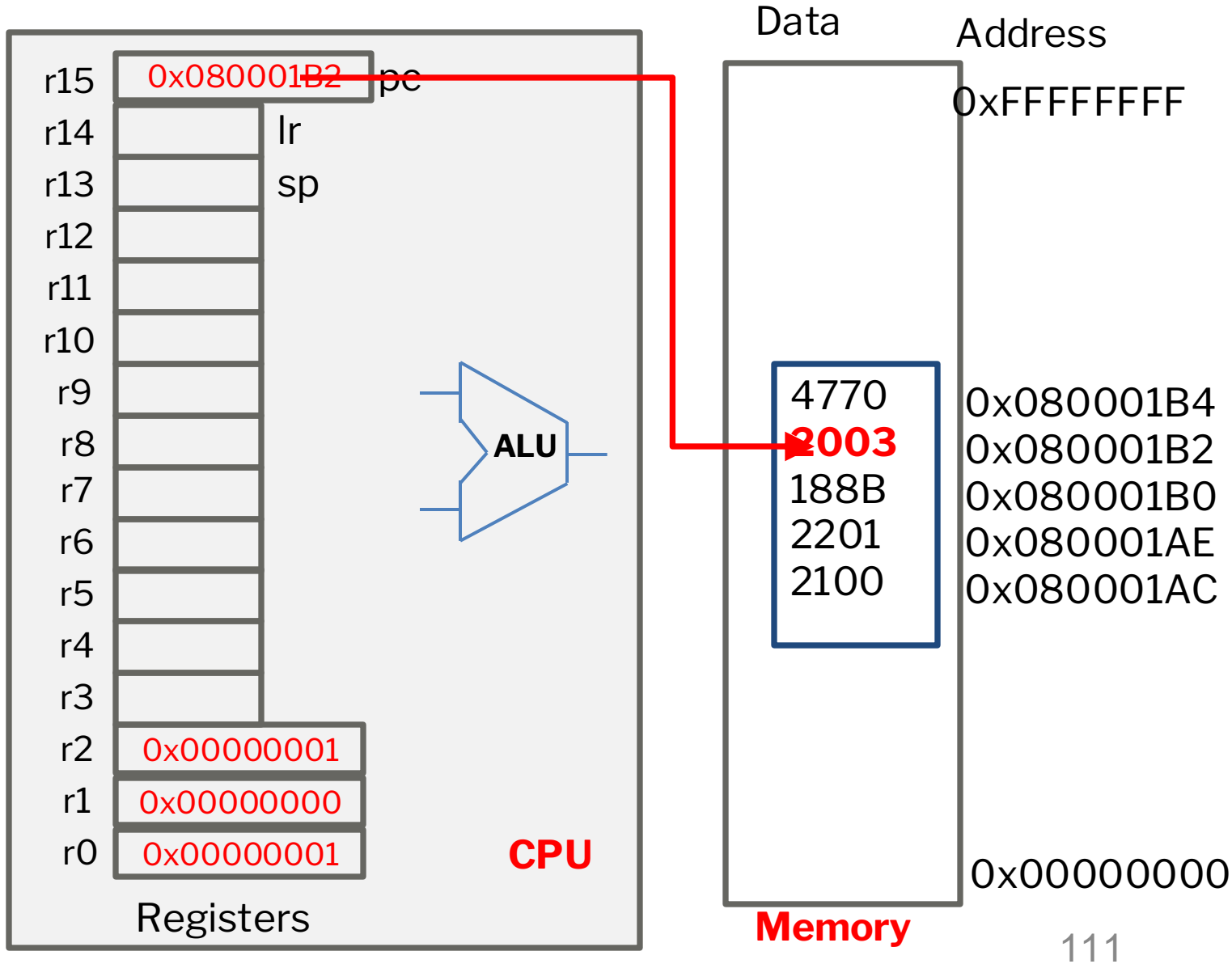
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2201 = MOVS r2, #0x01



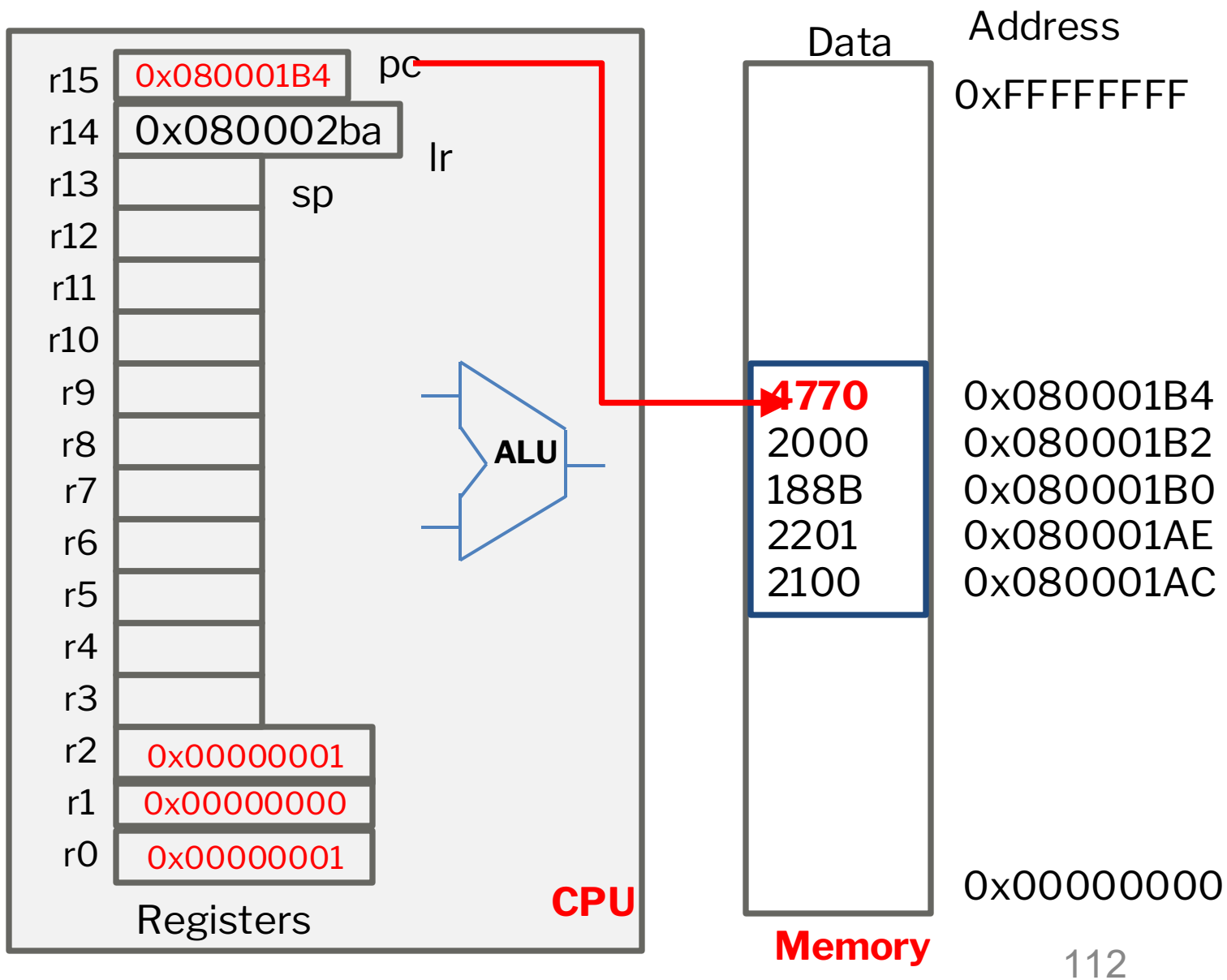
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 188B = ADDS r3, r1, r2



Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2000 = MOVS r0, r3



Fetch Next Instruction: pc = pc + 2
Decode & Decode: 4770 = BX lr

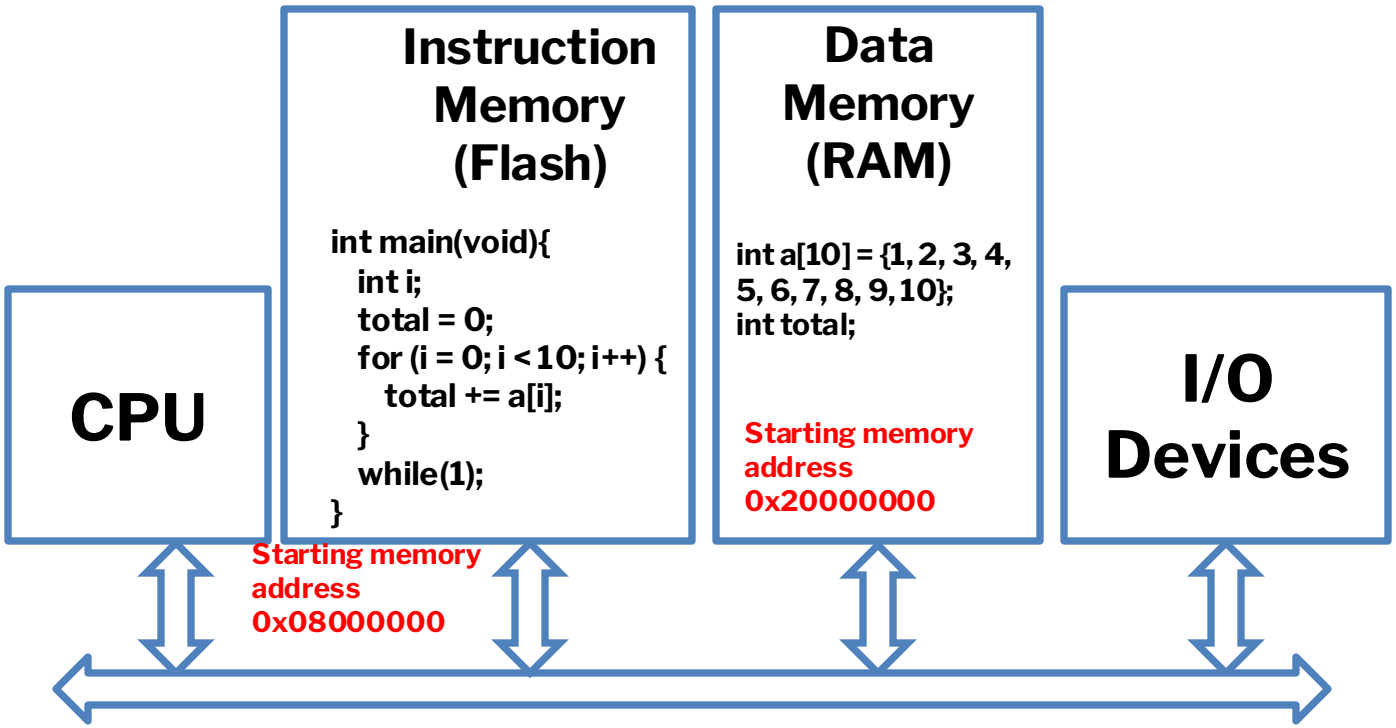


Example: Calculate the Sum of an Array

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;

int main(void){
        int i;
        total = 0;
        for (i = 0; i < 10; i++) {
      total += a[i];
        }
        while(1);
}
```

Example: Calculate the Sum of an Array



Example: Calculate the Sum of an Array

Instruction Memory (Flash)

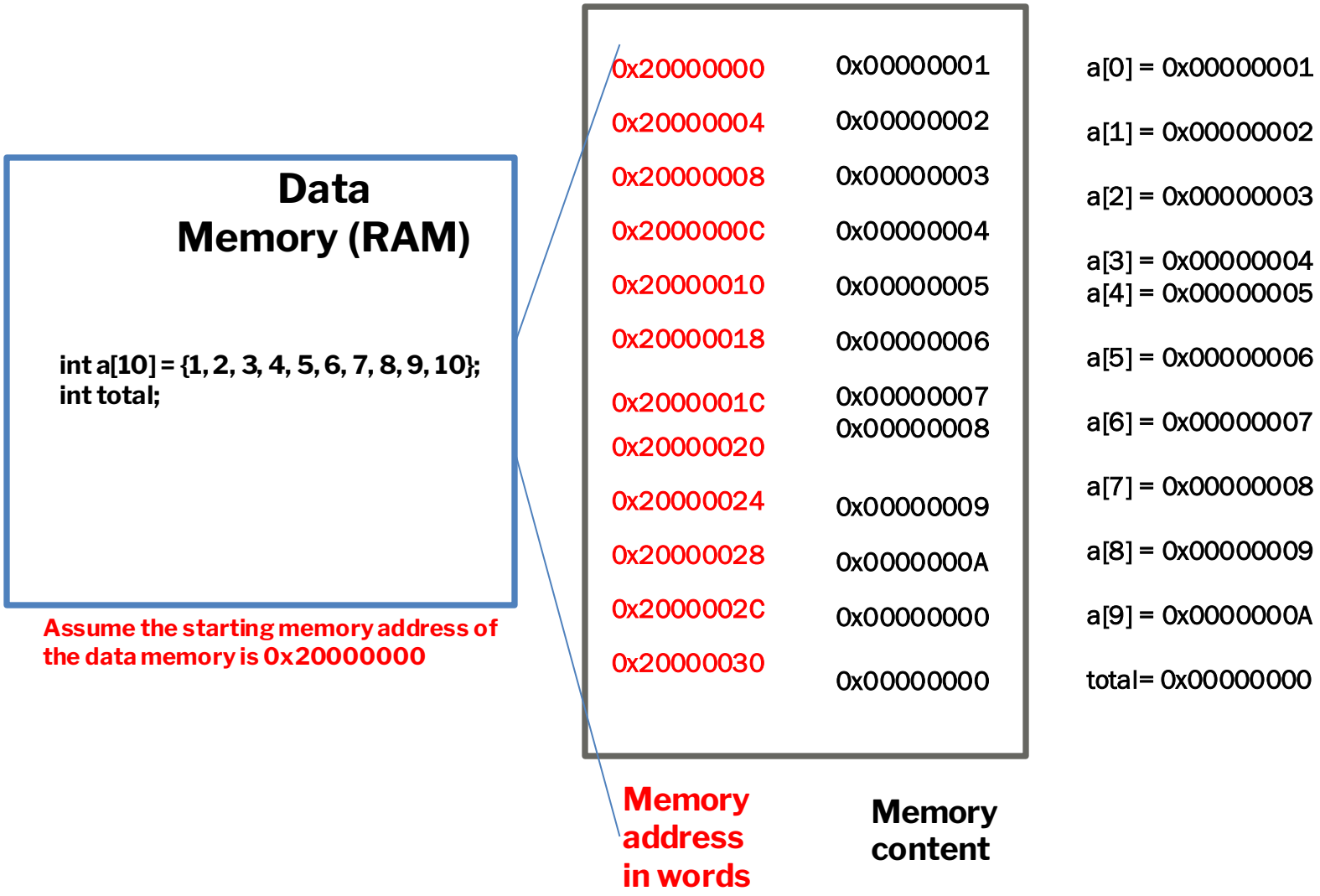
```
int main(void){
  int i;
  total = 0;
  for (i = 0; i < 10;
i++) {
     total += a[i];
  }
  while(1);
}
```

Starting memory address 0x08000000

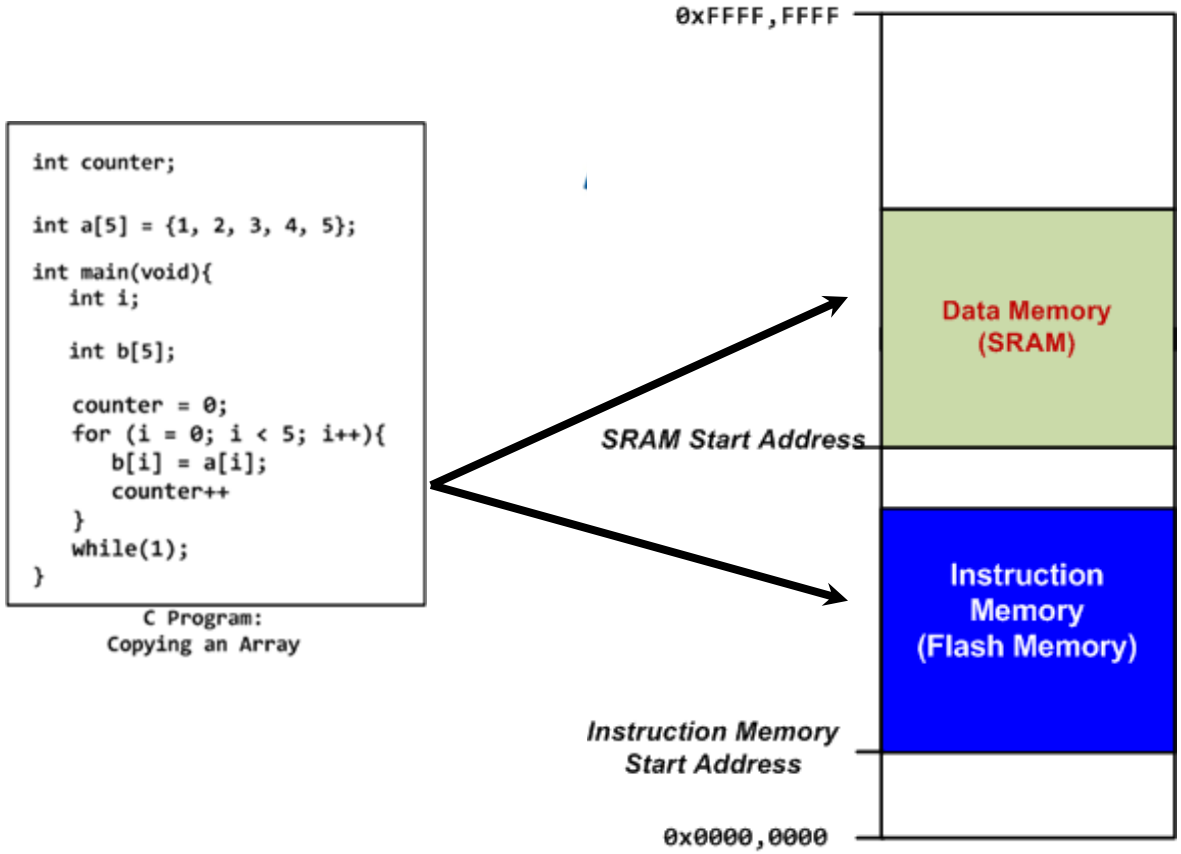
```
0010 0001 0000
0000
0100 1010 0000
1000
0110 0000 0001
0001
0010 0000 0000
0000
1110 0000 0000
1000
0100 1001 0000
0111
1111 1000 0101
0001
0001 0000 0010
0000
0100 1010 0000
0100
0110 1000 0001
0010
0100 0100 0001
0001
0100 1010 0000
0011
0110 0000 0001
0001
```

```
        MOVS  r1, #0x00
        LDR   r2, = total_addr
        STR   r1, [r2, #0x00]
        MOVS  r0, #0x00
        B     Check
Loop:   LDR   r1, = a_addr
        LDR   r1, [r1, r0, LSL #2]
        LDR   r2, = total_addr
        LDR   r2, [r2, #0x00]
        ADD   r1, r1, r2
        LDR   r2, = total_addr
        STR   r1, [r2,#0x00]
        ADDS  r0, r0, #1
Check:  CMP   r0, #0x0A
        BLT   Loop
        NOP
Self:   B     Self
```

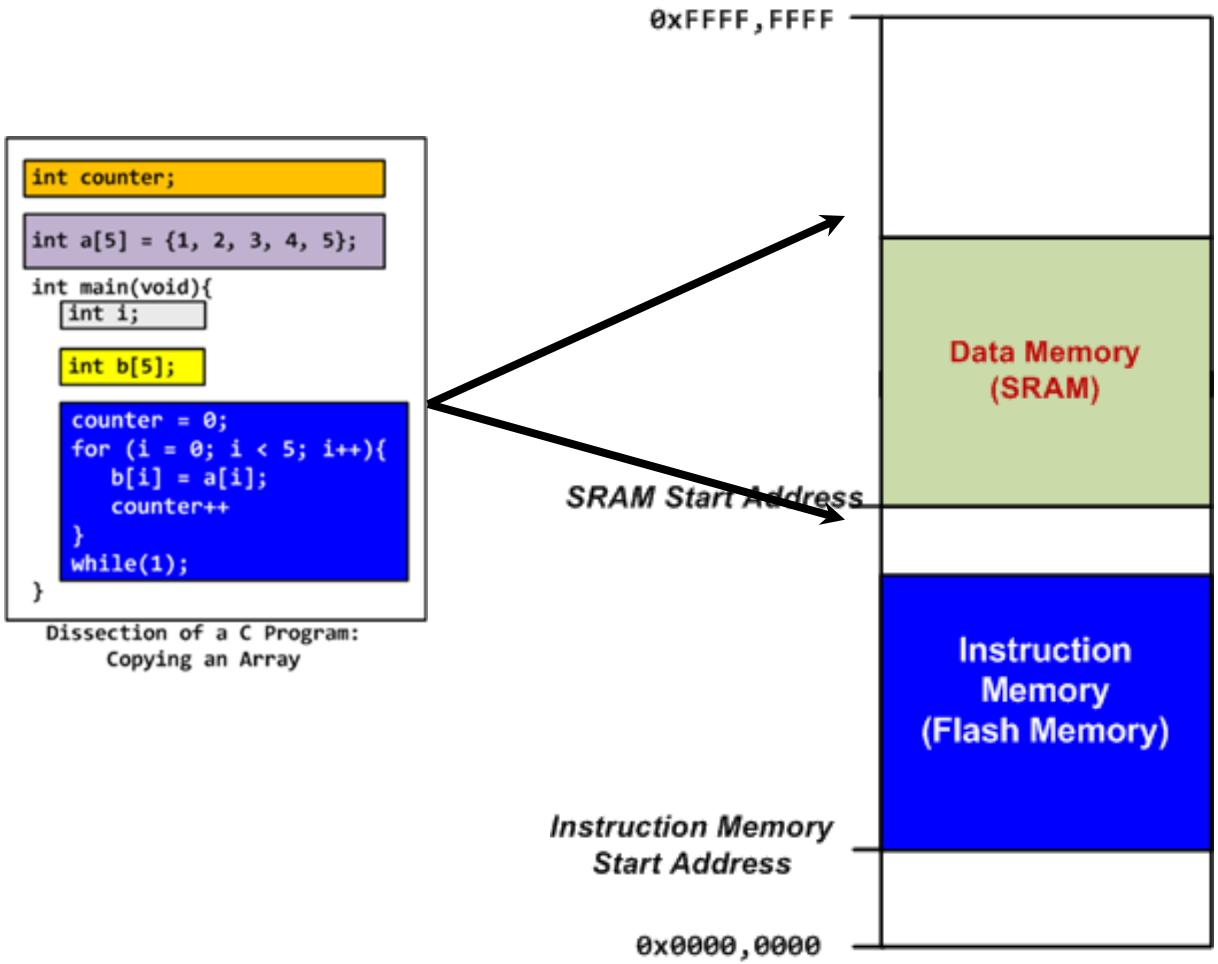
Example: Calculate the Sum of an Array



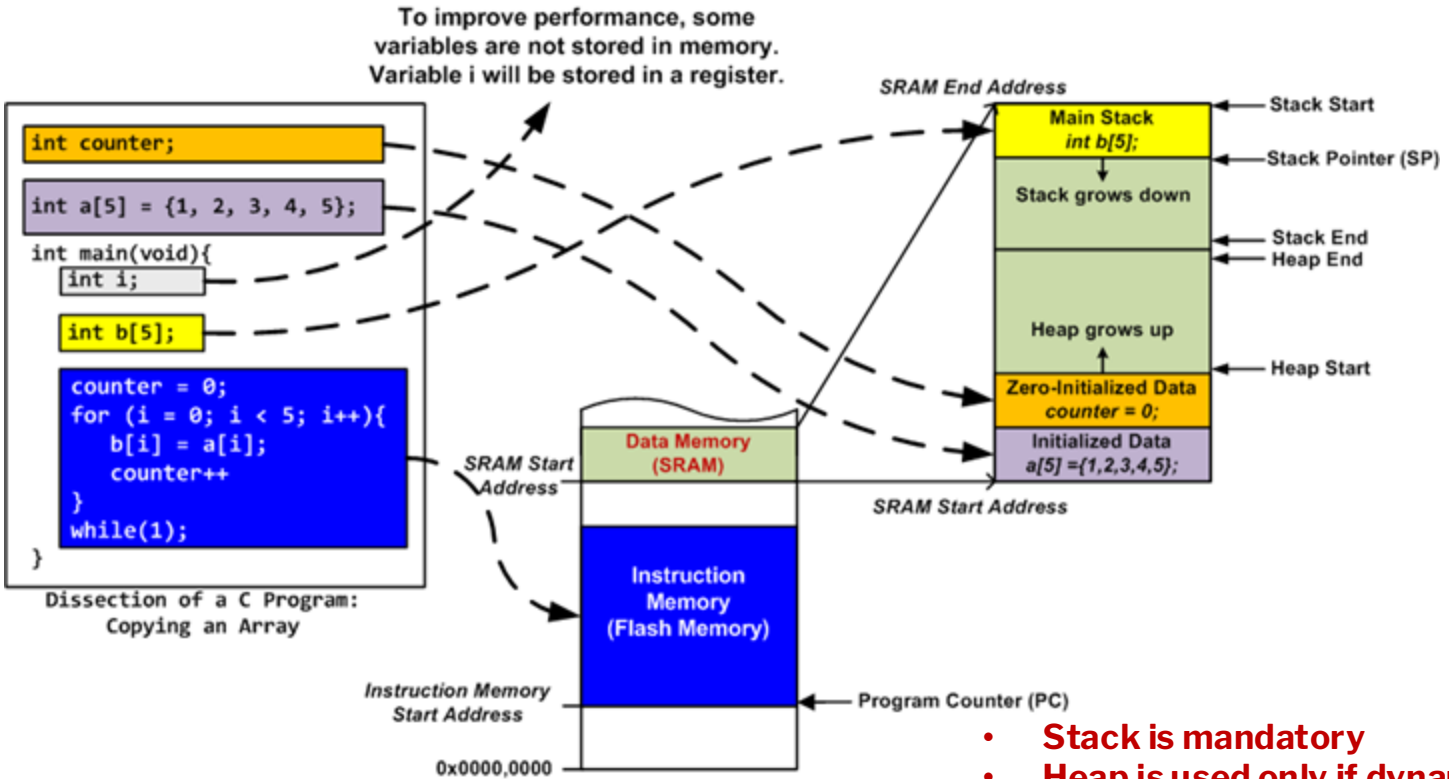
Loading Code and Data into Memory



Loading Code and Data into Memory



Loading Code and Data into Memory



- **Stack is mandatory**
- **Heap is used only if dynamic allocation (e.g. *malloc*, *calloc*) is used.**