CSIS 212 : Machine Organization & Assembly Language
Lecture 4

Duy Nguyen, Ph.D.

dnguyen@palomar.edu

858.204.5232 (cell)

Adder Circuit

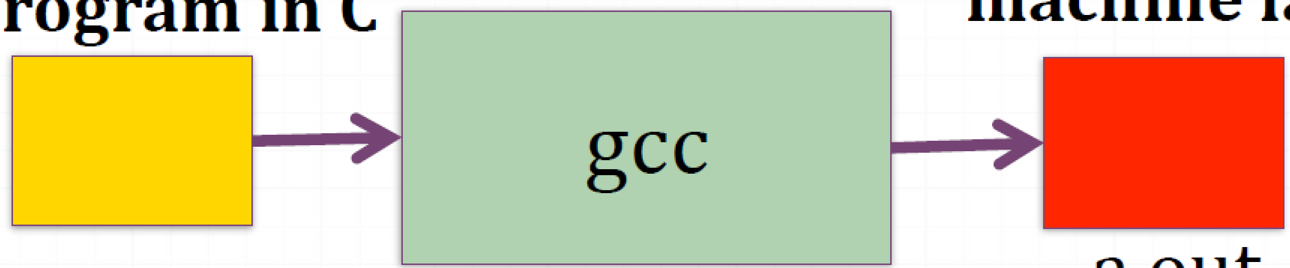
- **Half Adder**
- **Full Adder**
- **Rippler-Carry Adder Circuit**

What Does “gcc” Do?

% gcc hello.c

“Source” **Program in C**

“Executable”: Equivalent program in machine language



hello.c

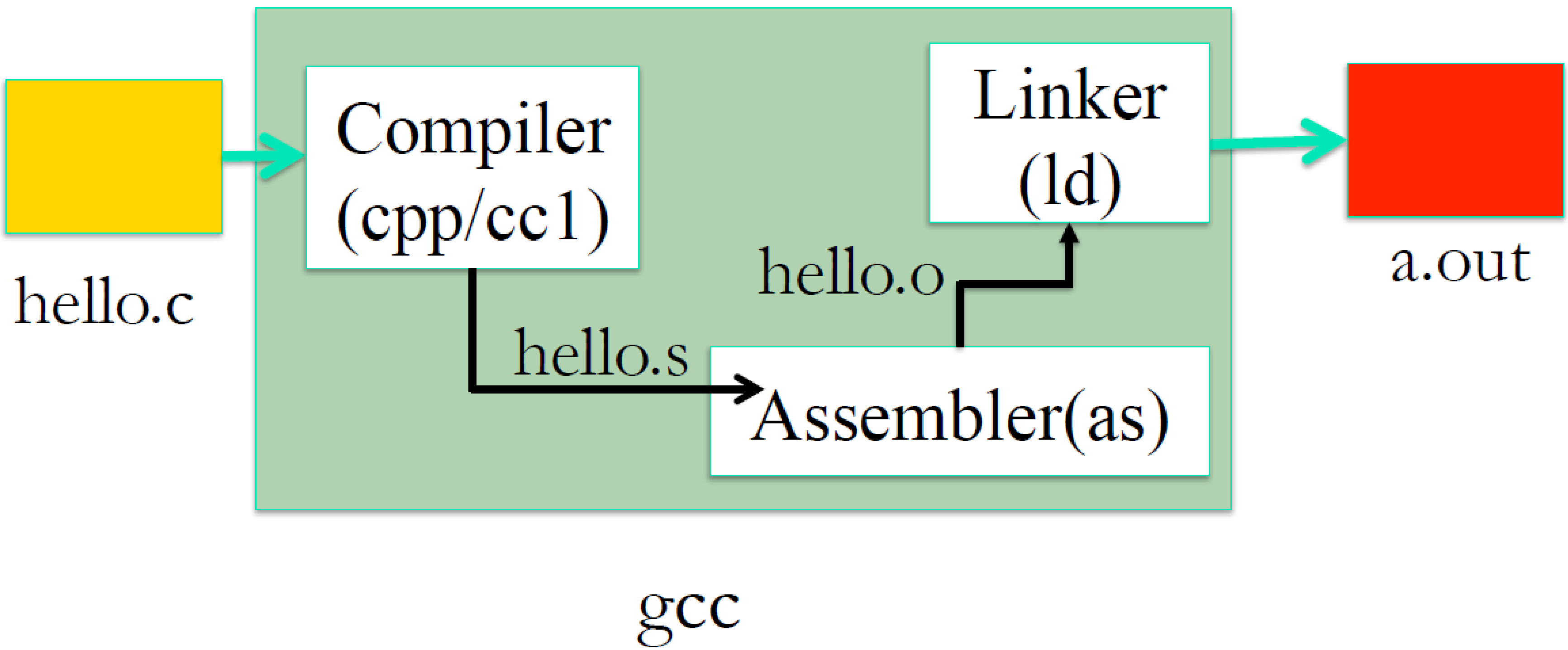
a.out

```
#include <stdio.h>
void func1(int a, char *b)
{
   if(a > 0)
   { *b = ‘a’ ;}
}
int main()
{.....
  func1();
   printf(“\abc”);
}
```

```
0000 1001 1100 0110
1010 1111 0101 1000
1010 1111 0101 1000
0000 1001 1100 0110
1100 0110 1010 1111
0101 1000 0000 1001
0101 1000 0000 1001
1100 0110 1010 1111
```

Steps in gcc

- **The translation is done in a number of steps**



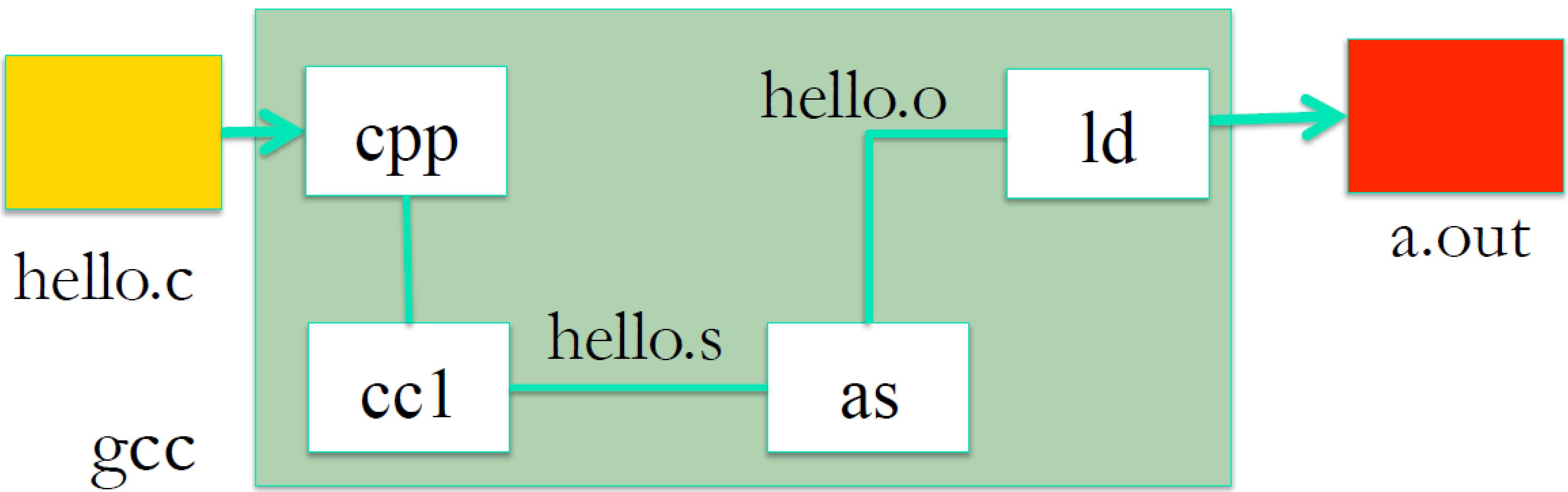
Steps in gcc

- Ask compiler to show temporary files:

% gcc –S hello.c   (gives hello.s – assembly code)

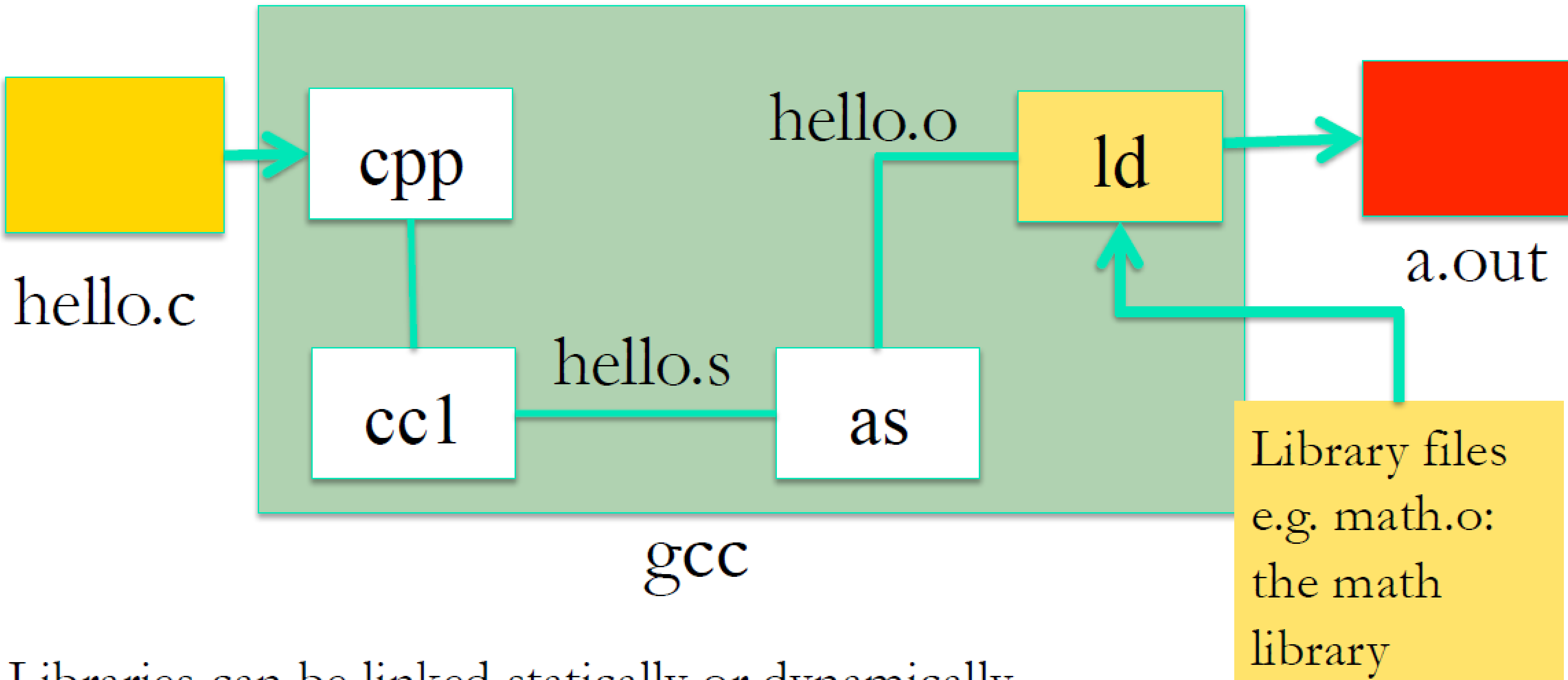
% gcc –c hello.c   (gives hello.o – object module)

% gcc –o prog_hello hello.c (gives prog_hello.o - named executable)



Include Code Written By Others

- Code written by others (libraries) can be included
- ld (linkage editor) merges one or more object files with the relevant libraries to produce a single executable



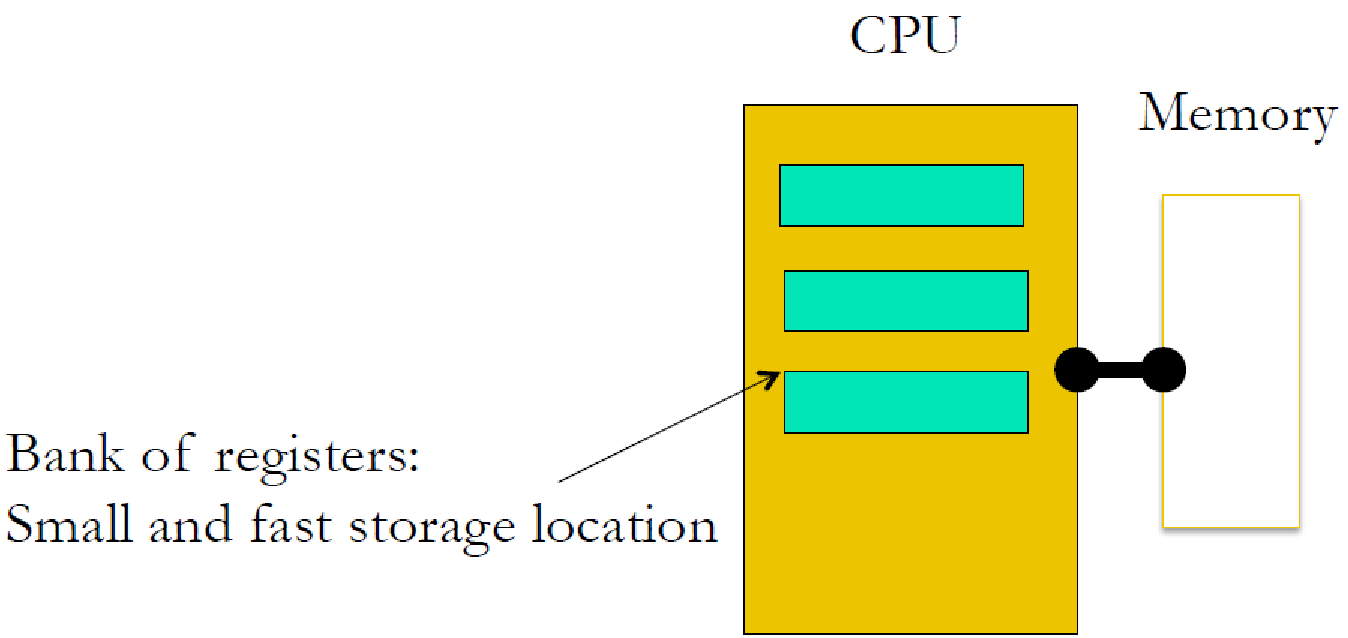
Libraries can be linked statically or dynamically

Machine vs. Assembly Language

- Machine Language: A particular set of instructions that the CPU can directly execute – but these are ones and zeros
- Assembly language is a symbolic version of the equivalent machine language
  - each statement (called an Instruction), executes exactly one of a short list of simple commands
  - Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
  - Instructions are related to operations (e.g. =, +, -, *) in C or Java

What is an Instruction Set Architecture?

1. Everything about h/w that is visible to the s/w and can be manipulated by it via basic machine instructions (System state)

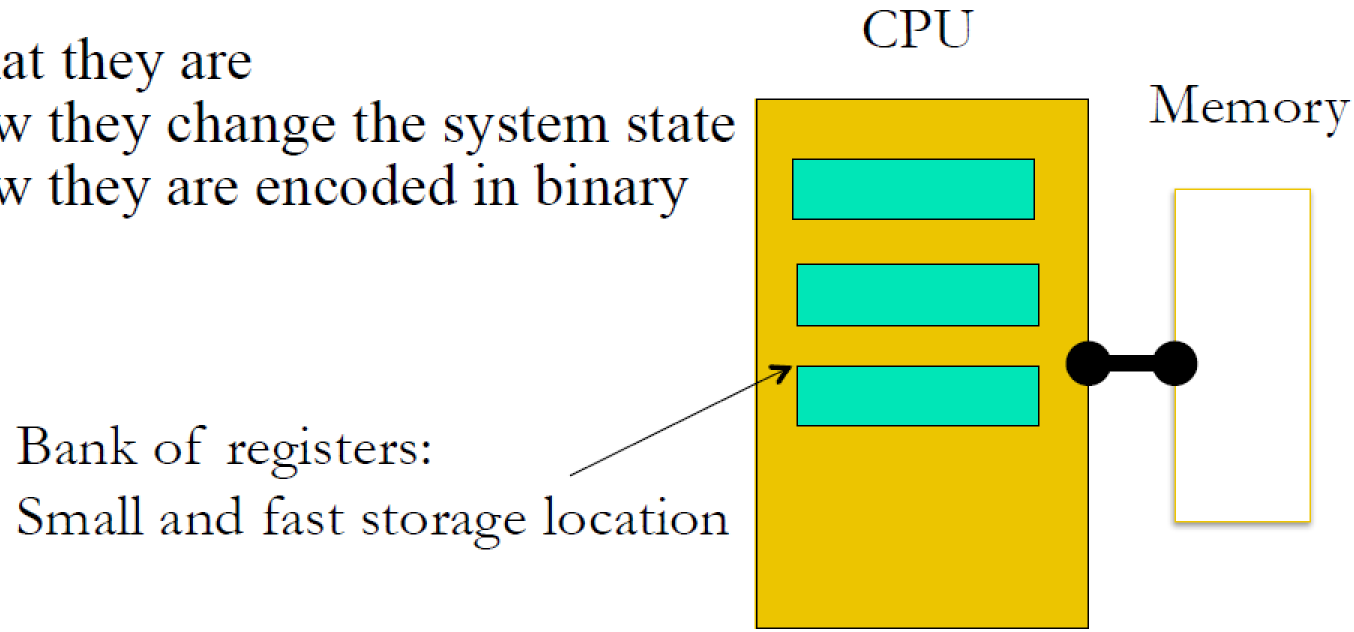


What is an Instruction Set Architecture

1. Everything about h/w that is visible to the s/w and can be manipulated by it via basic machine instructions (System state)
   Example: Registers: How many? What size?
   Memory: How to access contents?

2. The set of basic machine instructions:

   A. What they are
   B. How they change the system state
   C. How they are encoded in binary



Is the ISA Different for Different CPUs?

- Different CPUs implement different sets of instructions.
  - Examples: ARM, Intel x86, IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA32 …
- Two styles of CPU design:
  - RISC (Reduced Instruction Set Computing)
  - CISC (Complex Instruction Set Computing)

RISC vs. CISC

- Complex Instruction Set Computing e.g x86
  - Larger instruction set
  - More complicated instructions built into hardware
  - Variable length
  - Multiple clock cycles per instruction

- Reduced Instruction Set Computing e.g. ARM
  - Small, highly optimized set of instructions
  - Memory accesses are specific instructions
  - One instruction per clock cycle
  - Instructions are of the same size and fixed format

A = A*B

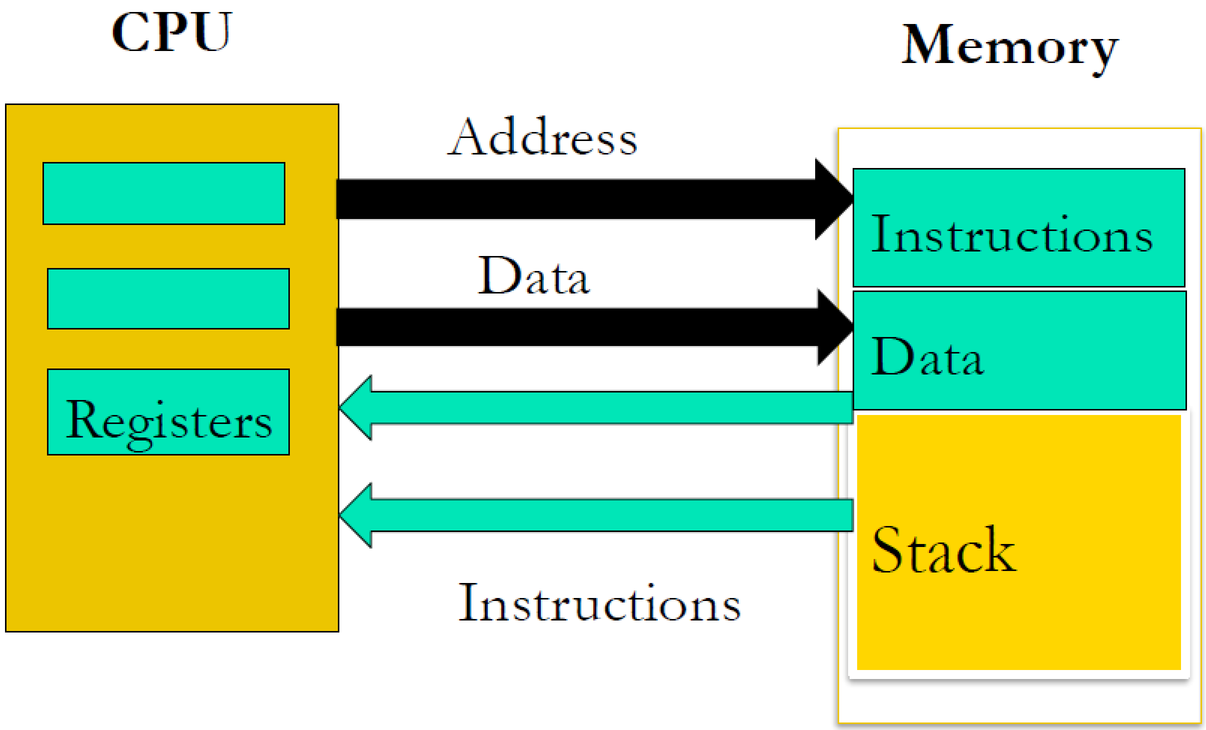
RISC

```
LOAD A, eax
LOAD B, ebx
PROD eax, ebx
STORE ebx, A
```

CISC

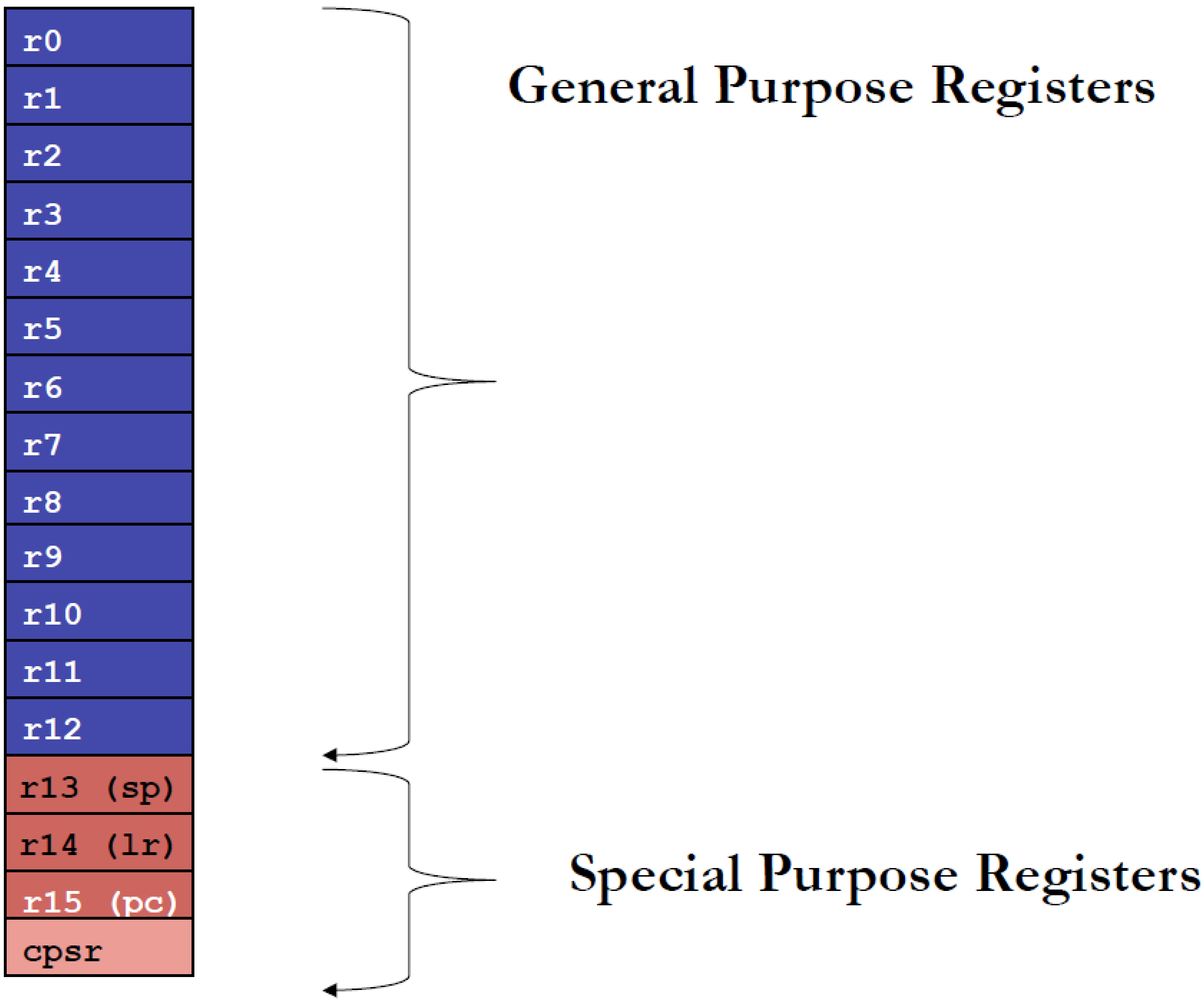
```
MULT B, A
```

The Assembly Programmer's View of the Machine



- Registers: (Very) Small amount of memory inside the CPU
- Each ARM register is 32 bits wide
  - Groups of 32 bits called a word in ARM

The ARM Register Set



ARM Assembly Variables: Registers

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Data is put into a register before it is used for arithmetic, tested, etc.
- Manipulated data is then stored back in main memory.
- Benefit: Since registers are directly in hardware, they are very fast

C, Java Variables vs. Registers

- In C (and most High Level Languages) variables declared first and given a type
  - Example:
    ```
    int fahr, celsius;
    char a, b, c, d, e;
    ```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated

Basic Types of ARM Instructions

1. Arithmetic: Only processor and registers involved
   1. compute the sum (or difference) of two registers, store the result in a register
   2. move the contents of one register to another

2. Data Transfer Instructions: Interacts with memory
   1. load a word from memory into a register
   2. store the contents of a register into a memory word

3. Control Transfer Instructions: Change flow of execution
   1. jump to another instruction
   2. conditional jump (e.g., branch if registeri == 0)
   3. jump to a subroutine

ARM Addition and Subtraction

- Syntax of Instructions:

  1 2, 3, 4

  where:

  1) instruction by name

  2) operand getting result (“destination”)

  3) 1st operand for operation (“source1”)

  4) 2nd operand for operation (“source2”)

- Syntax is rigid (for the most part):
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

Addition and Subtraction of Integers

- Addition in Assembly
  - Example: `ADD r0,r1,r2` (in ARM)

    Equivalent to: `a = b + c` (in C)

    where ARM registers `r0,r1,r2` are associated with C variables `a, b, c`
- Subtraction in Assembly
  - Example: `SUB r3, r4, r5` (in ARM)

    Equivalent to: `d = e - f` (in C)

    where ARM registers `r3,r4,r5` are associated with C variables `d, e, f`

Setting Condition Bits

- Simply add an 'S' following the arithmetic/ logic instruction
  - Example: `ADDS r0,r1,r2` (in ARM)

  This is equivalent to r0=r1+r2 and set the condition bits for this operation

What is the Minimum Number of Assembly Instructions Needed to Perform the Following?

```
a = b + c + d - e;
```

A. Single instruction
B. Two instructions
C. Three instructions
D. Four instructions

Assume the value of each variable is stored in a register.

Addition and Subtraction of Integers

- How do the following C statement?

```
a = b + c + d - e;
```

- Break into multiple instructions

```
ADD r0, r1, r2     ; a = b + c
ADD r0, r0, r3     ; a = a + d
SUB r0, r0, r4     ; a = a - e
```

- Notice: A single line of C may break up into several lines of ARM.
- Notice: Everything after the semicolon on each line is ignored (comments)

Addition and Subtraction of Integers

- How do we do this?
  - `f = (g + h) - (i + j);`
- Use intermediate temporary register

```
ADD r0,r1,r2          ; f = g + h
ADD r5,r3,r4          ; temp = i + j
SUB r0,r0,r5          ; f =(g+h)-(i+j)
```

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are ways to indicate their existence
- Add Immediate:
  - `f = g + 10` (in C)
  - `ADD r0,r1,#10` (in ARM)
  - where ARM registers `r0,r1` are associated with C variables `f, g`
- Syntax similar to `add` instruction, except that last argument is a #number instead of a register.

Arithmetic Operations: Addressing Modes

1. Register Direct Addressing: Operand values are in registers:
   - ADD r3, r0, r1; r3=r0+r1
2. Immediate Addressing Mode: Operand value is within the instruction
   - ADD r3, r0, #7; r3=r0+7
   - The number 7 is stored as part of the instruction
3. Register direct with shift or rotate (more next lecture)
   - ADD r3, r0, r1, LSL#2; r3=r0+ r1<<2

Add/Subtract Instructions

1. ADD r1, r2, r3; r1=r2+r3
2. ADC r1, r2, r3; r1=r2+r3+ C(arry Flag)
3. SUB r1, r2,r3; r1=r2-r3
4. SUBC r1, r2, r3; r1=r2-r3 +C -1
5. RSB r1, r2, r3; r1= r3-r2;
6. RSC r1, r2, r3; r1=r3-r2 +C -1