SoC 101:

*a.k.a., "**Everything you wanted to know about a computer but were afraid to ask**"*

Lecture 3: From C to Assembly

Prof. Adam Teman

EnICS Labs, Bar-Ilan University

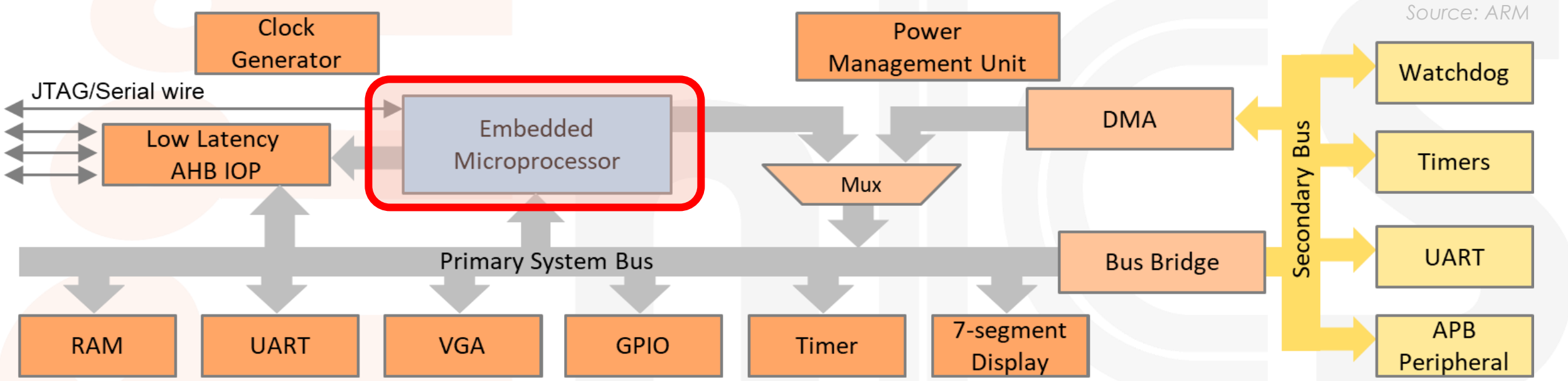
29 April 2023

Emerging Nanoscaled Integrated Circuits and Systems Labs

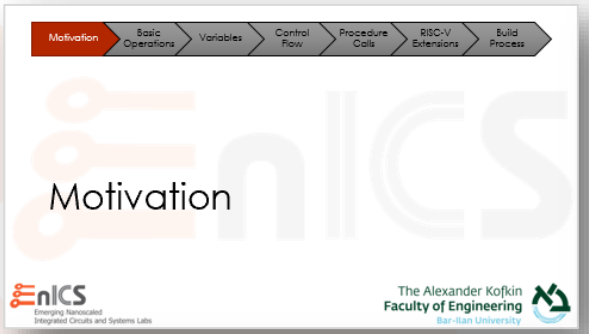
The Alexander Kofkin Faculty of Engineering

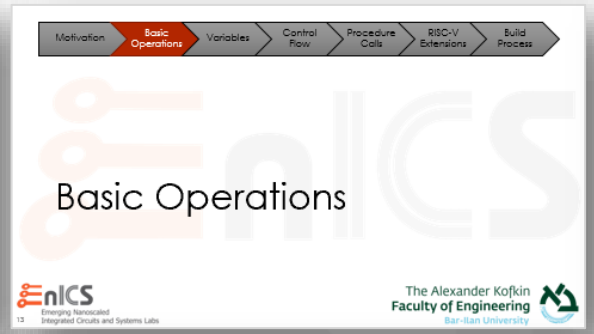
Bar-Ilan University

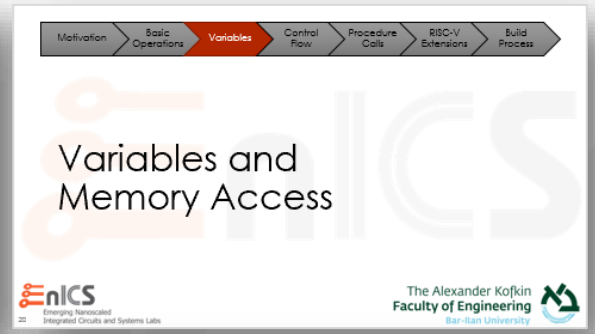
This Lecture

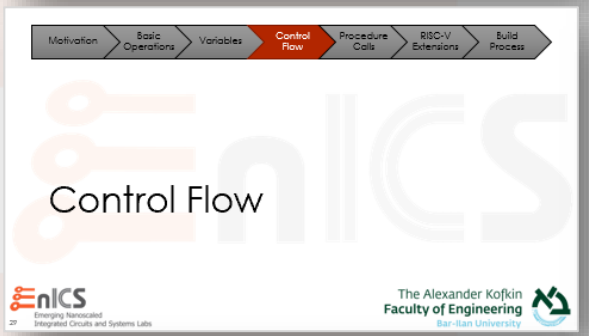


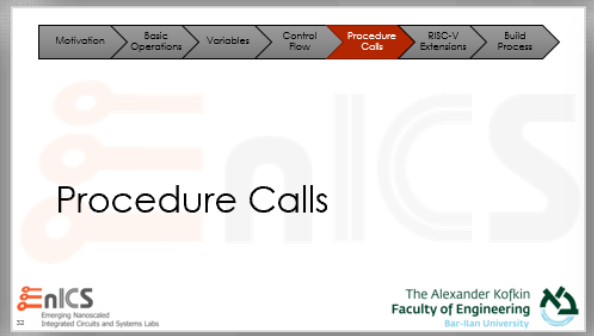
Outline

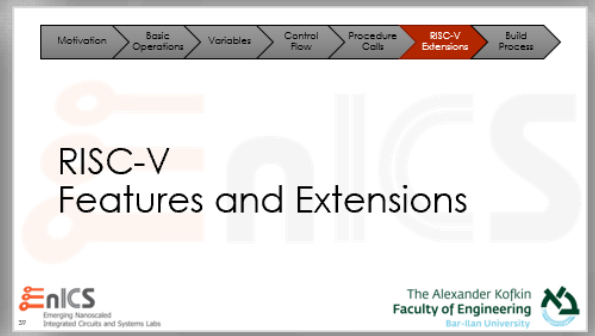






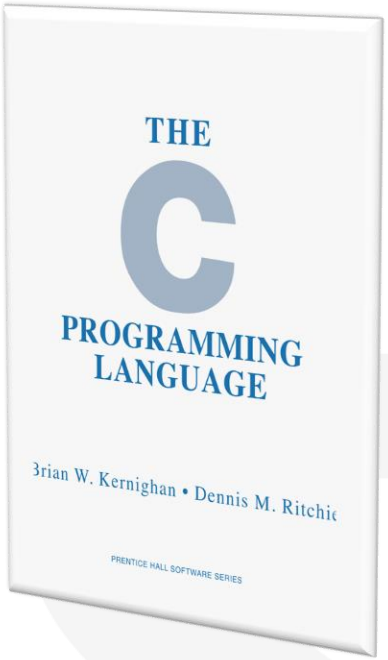






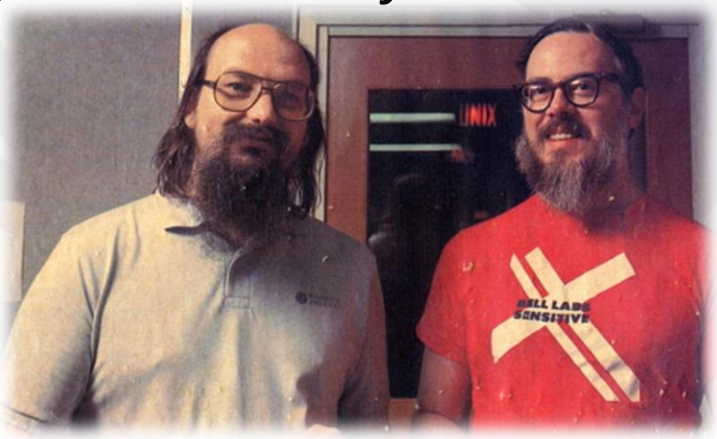
Motivation

The ‘C’ Programming Language



- **Fifty years and counting…**
  - C was developed in the early 70s by Dennis Ritchie and Ken Thompson.
    * ‘C’ replaced ‘B’, which was named for Ken Thompson’s wife, Bonnie.
  - “ANSI C” (C89) is often considered the standard.
  - Today, C is still the preferred language for programming embedded systems.
- **Why?**

  For many reasons, but here are a few of the main ones:
  - Fine-grained Control
  - Memory Management
  - Performance
  - Bit Manipulation
  - Portability and Compatibility

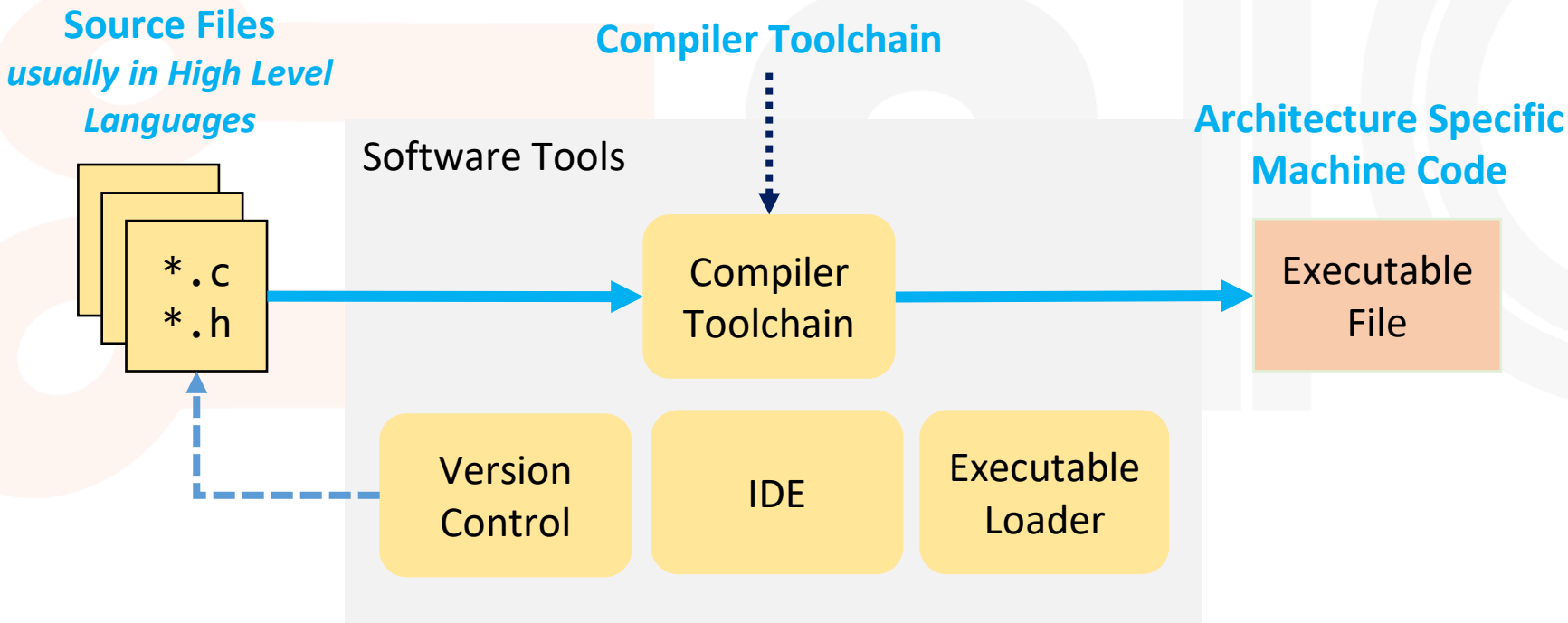


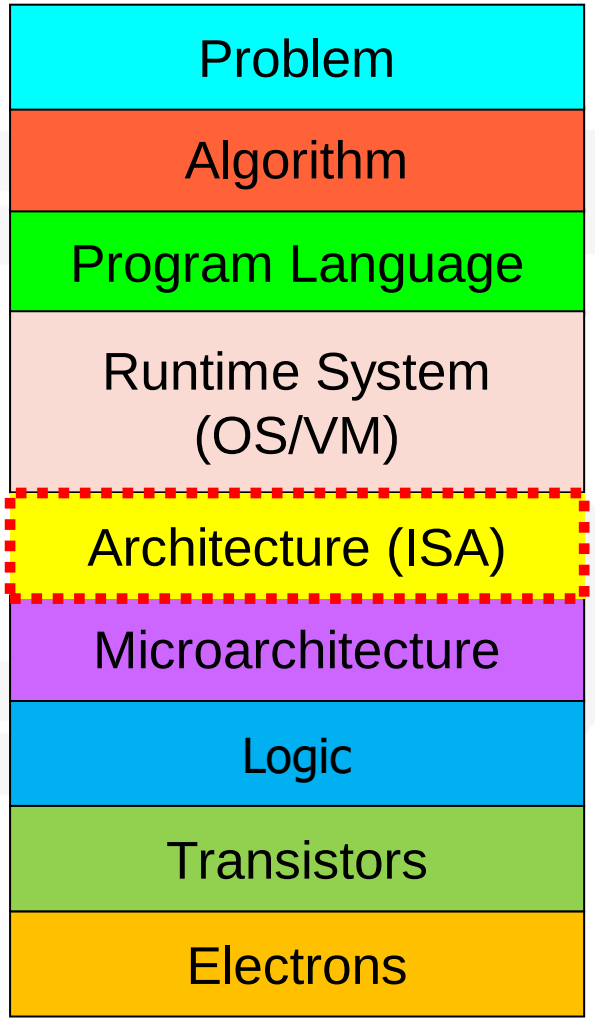
Source: computerhistory.org

“C is the closest thing to assembly that is not assembly.”

But hardware runs on binaries

- **The Instruction Set Architecture (ISA) is the interface ("contract") between the software and the hardware.**



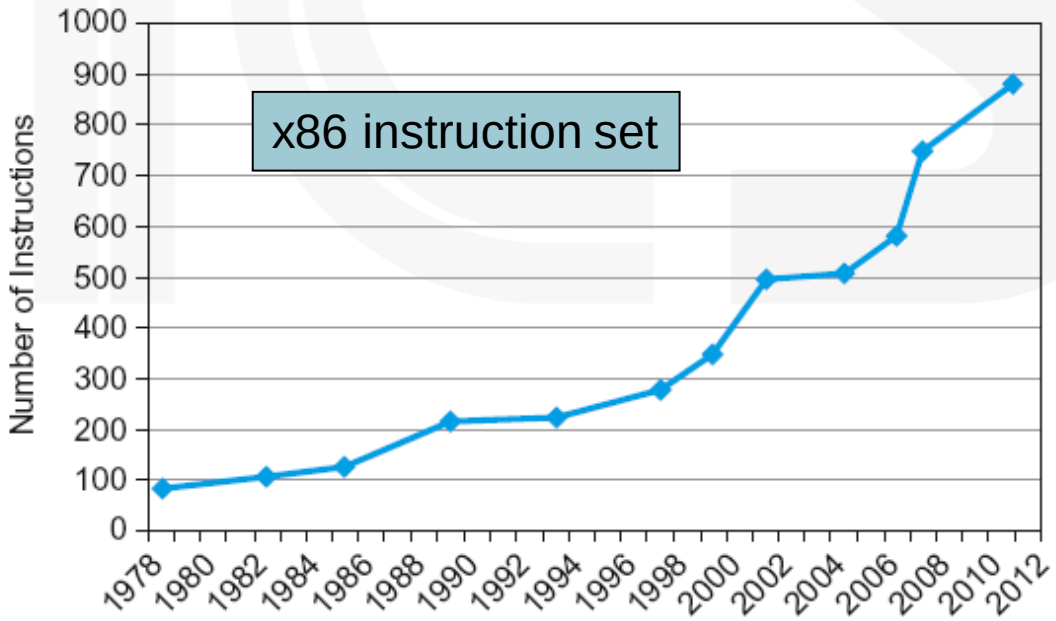


Why Instruction Set Architecture matters

- **Why can't Intel sell mobile chips?**
  - 99%+ of mobile phones/tablets based on ARM v7/v8/v9 ISA
- **Why can't ARM partners sell servers?**
  - 99%+ of laptops/desktops/servers based on AMD64 (x86-64) ISA
- **How can IBM still sell mainframes?**
  - IBM 360, oldest surviving ISA (50+ years)

- **Instruction Sets do not change**
  - But they do accrete more instructions

ISA is most important interface in computer system where software meets hardware



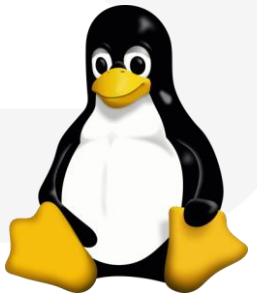
Source: P&H, Ch. 2

Proprietary ISAs Die Out

- **Proprietary ISA** fortunes tied to business fortunes and whims

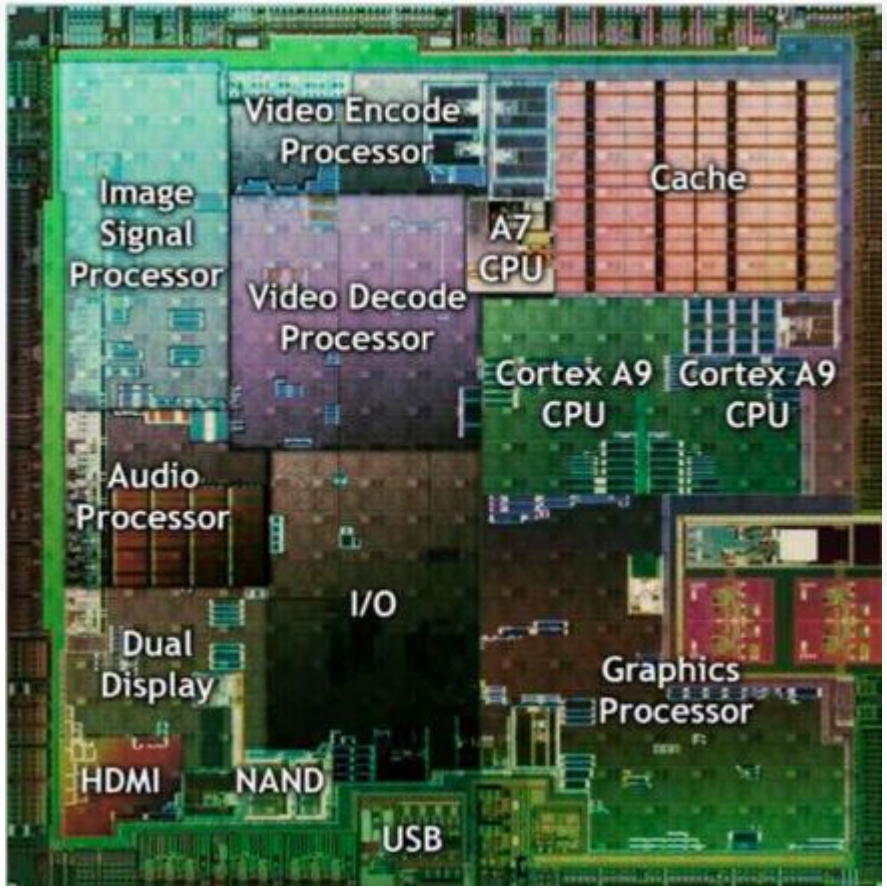
- **Open Interfaces work for Software. Why not for Hardware?!?**

Field	Open Standard	Proprietary Implemen.	Free, Open Implementation
Networking	Ethernet, TCP/IP	Many	Many
OS	Posix	MS Windows	Linux, FreeBSD
Compilers	C	Intel icc, ARMcc	gcc, LLVM
Databases	SQL	Oracle 12C, MS DB2	MySQL, PostgresSQL
Graphics	OpenGL	MS DirectX	Mesa3D
ISA	????	x86, ARM, IBM360	------



The Need for a Single ISA

- **Modern SoCs have many different ISAs on a single SoC, such as:**
  - Applications processor (usually ARM)
  - Graphics processors
  - Image processors
  - Radio DSPs
  - Audio DSPs
  - Security processors
  - Power-management processor
- **A Single ISA is invaluable**
  - A single software stack
  - No proprietary ISAs that may disappear
  - Flexibility for various needs and features



NVIDIA Tegra SoC

Source: NVIDIA

The solution: RISC-V

- **Summer 2010: "3-month project" at UC-Berkeley**
  - Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic
- **May 2014: Frozen Base User Spec**
- **2015: RISC-V Foundation Established**
  - Led by Calista Redmond since 2019
  - Over 3100 members (2023)
- **RISC-V Project Goal:**

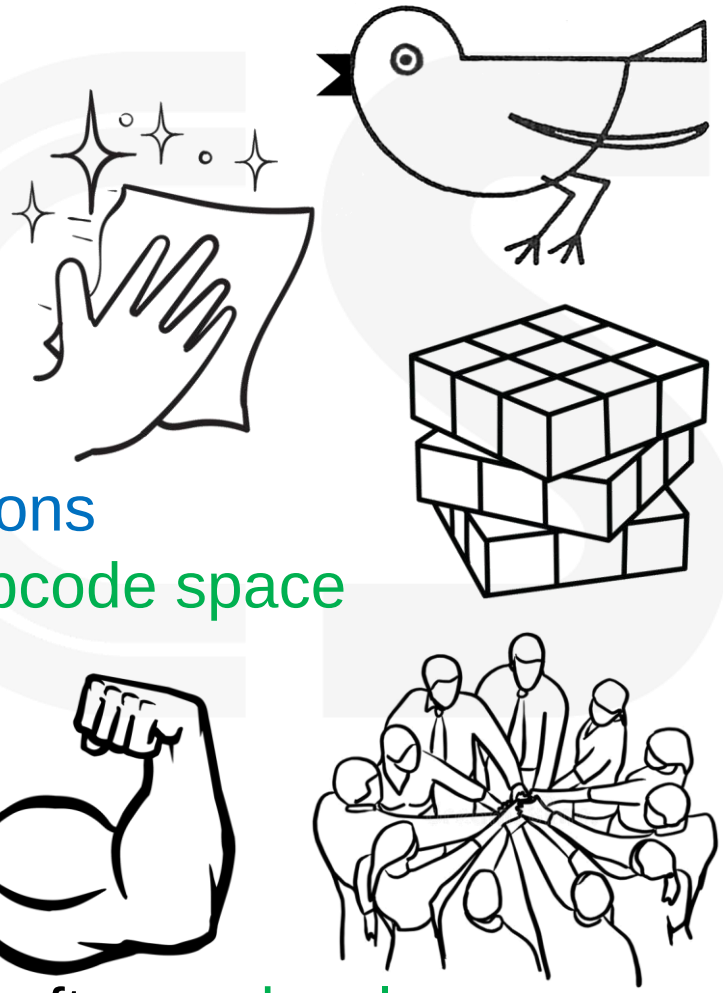
Become the industry-standard ISA for all computing devices

Source: SiFive

Source: K. Asanovic

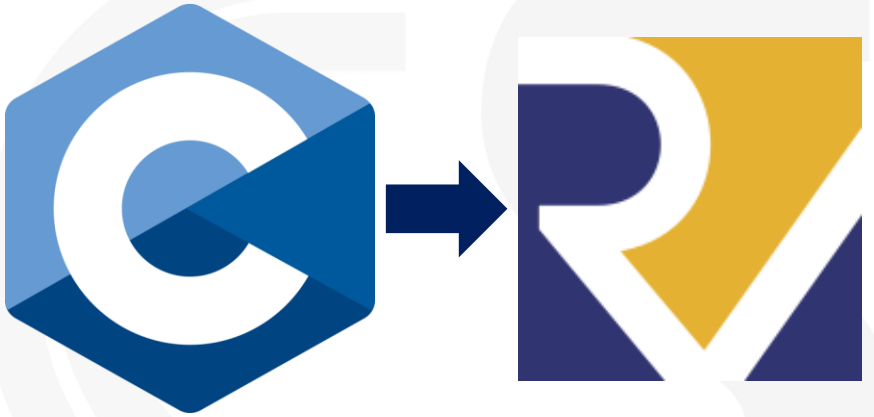
What's Different about RISC-V?

- **Simple**
  - Far smaller than other commercial ISAs
- **Clean-slate design**
  - Clear separation between user and privileged ISA
  - Avoids μarchitecture or technology-dependent features
- **Modular ISA designed for extensibility/specialization**
  - Small standard base ISA, with multiple standard extensions
  - Sparse & variable-length instruction encoding for vast opcode space
- **Stable**
  - Base and first standard extensions are frozen
  - Additions via optional extensions, not new versions
- **Community designed**
  - Developed with leading industry/academic experts and software developers



From Software to Hardware

- **In this lecture, we will cross the boundary between software and hardware.**
  - At the software side, we will look at C as a high-level programming language.
  - At the hardware side, we will use the RISC-V ISA to demonstrate assembly and machine language.
  - Specifically, we'll be using the 32-bit RISC-V integer instructions (RV32I).



- **This overview will show how high-level programming constructs map to the CPU architecture introduced in the previous lecture.**
  - However, these concepts are applicable to any programming language and any instruction set architecture.

Basic Operations

Emerging Nanoscaled
Integrated Circuits and Systems Labs

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Our basic computer

- **From the last lecture, our basic computer comprises:**
  - Control and Datapath
  - Program Counter
  - General Purpose Registers
  - Instruction and Data Memories
- **As a load-store architecture operations are done directly on registers, e.g.:**

```
add x1,x2,x3 → x1←x2+x3
```

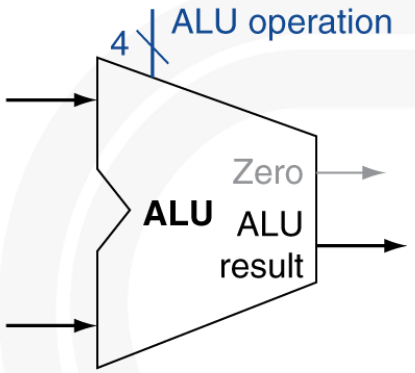
- **Such an operation has three components:**
  1. Instruction Fetch
  2. Register File Access
  3. ALU Execution

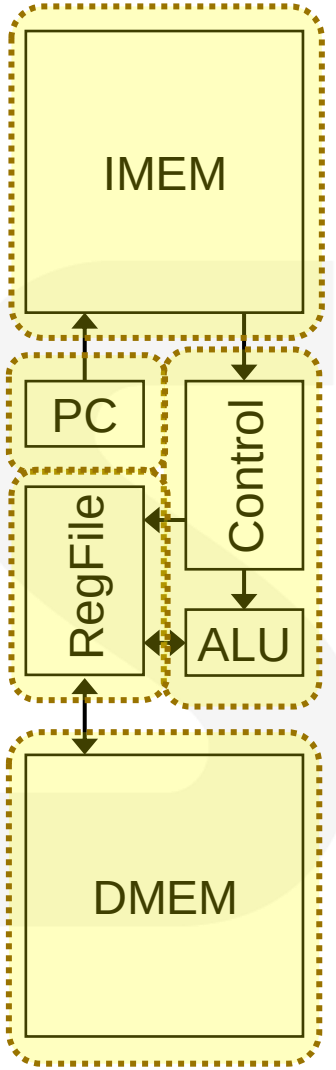
C Code:

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

RV ASM:

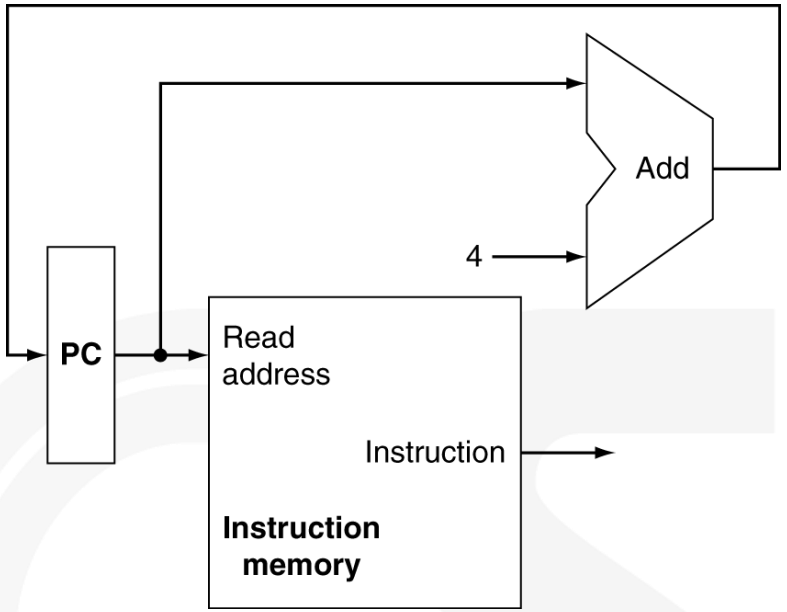
```
add x5,x1,x2
add x6,x3,x4
sub x7,x6,x5
```



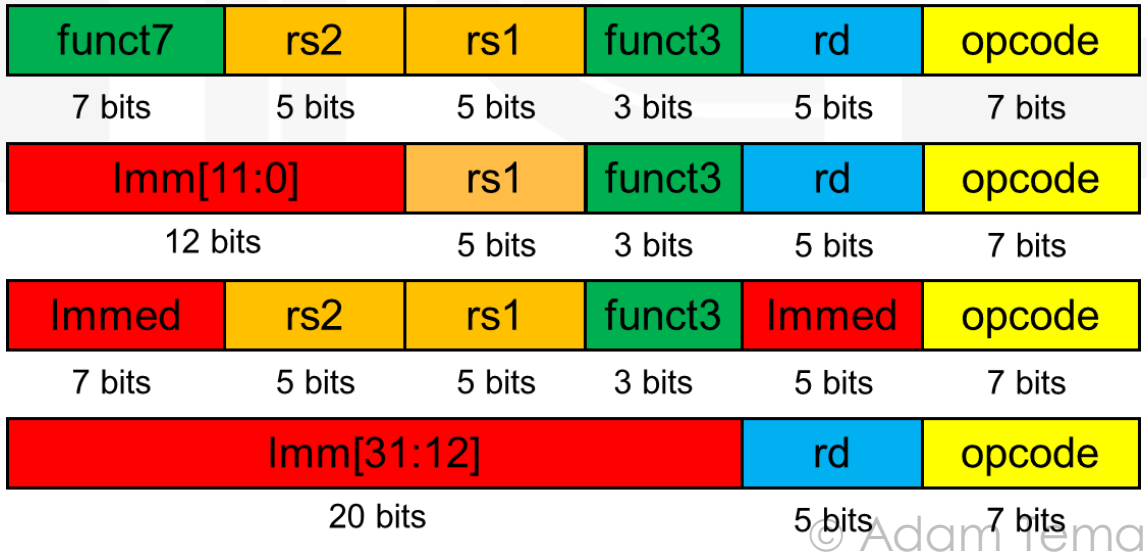


- ***Let us start building our datapath with these components***

1. Instruction Fetch



- **Instructions are 32 bits wide, so for every instruction:**
  - We need to fetch one 32-bit word
  - And increment the address by 4-bytes
- **Instructions come in a number of formats**
  - *Bit placement* optimized for hardware implementation
  - For example, all store the opcode in the bottom 7 bits
- **The instruction formats in RISC-V are:**
  - R (Register)-Format
    - 2-source, 1-destination operand
  - I (Immediate)-Format
    - 1-source, 1-destination, 12-bit constant
  - S (Store) and B (Branch)-Format
    - 2-source, 12-bit constant
  - U and J (Jump)-Format
    - 1-destination, 20-bit constant



2. Register File Access

source: elearningindustry.com

- **RISC-V has 32 Registers**
  - The "Goldilocks Principle":
  - *"This porridge is too hot; This porridge is too cold; This porridge is just right"*
  - Smaller is faster, but too small is bad.
- **Registers are numbered `x0` to `x31`**
  - Actually, it's 31 registers, since `x0` is hard-wired to `0`
  - All other registers are equivalent/general purpose
  - Actually, it's 32 registers, since there's also the program counter (`pc`)
- **The Application Binary Interface (ABI) gives certain registers assignments:**
  - `x1`: Return address (`ra`) `x2`: Stack pointer (`sp`) `x3`: Global pointer (`gp`)
    `x8`: Frame pointer (`fp`) `x10-11`: Return values `x10-17`: Arguments (e.g., `a0`)
    `x5-7,28-31`: temporaries (e.g., `t0`) `x8-9,18-27`: saved registers

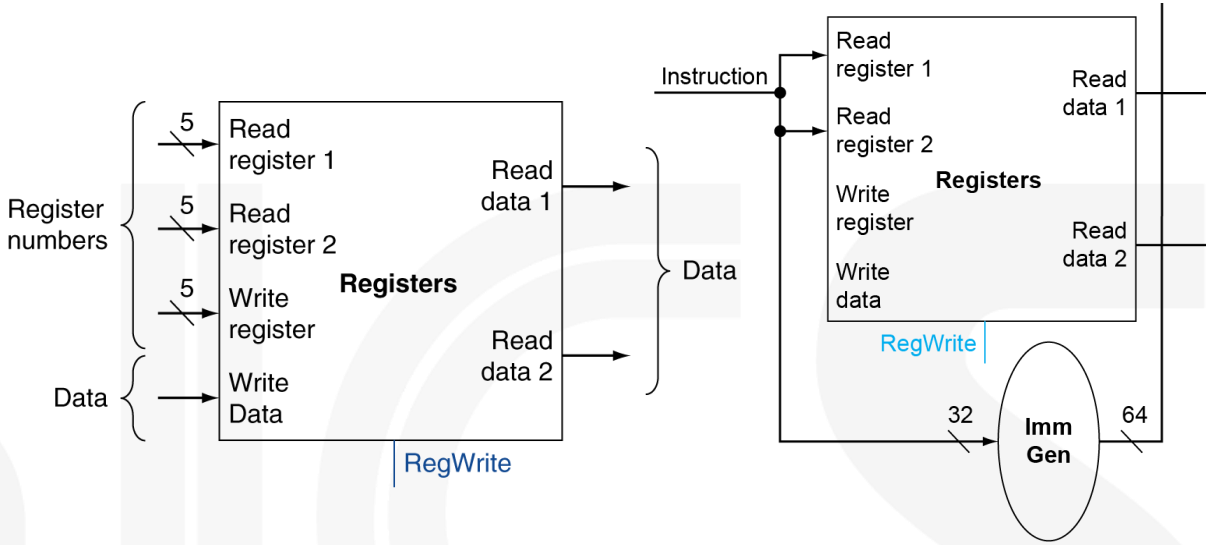
Reg.	ABI Name	Description
x0	zero	Hard-wired Zero
x1	ra	Return Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x5-7	t0-2	Temporaries
x8	s0/fp	Frame Pointer/ Saved Reg
x9	s1	Saved Register
x10-11	a0-1	Arguments/ Return Values
x12-17	a2-7	Arguments
x18-27	s2-11	Saved Registers
x28-31	t3-6	Temporaries

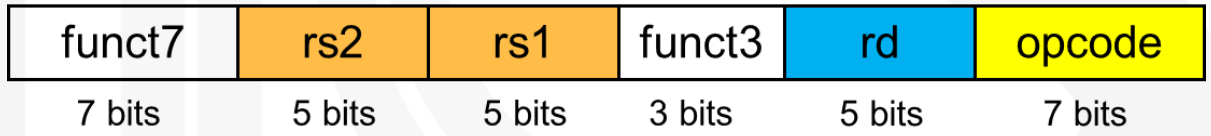
2. Register File Access

- **A Register-Register Operation requires:**

```
add x1,x2,x3 → x1←x2+x3
```

  - Two source operands: `rs1`, `rs2`
  - One destination operand: `rd`
  - → Register file requires 2R1W access
  - This operation will use the R-Format:

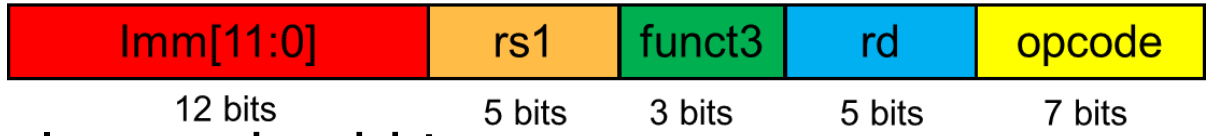




- **Arithmetic with a constant requires:**

```
addi x1,x2,0x123 → x1←x2+0x123
```

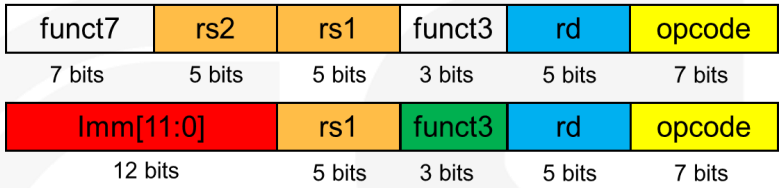
  - One source (`rs1`) and one destination (`rd`) operand.
  - An "immediate" in the remaining available bits.
  - This operation will use the I-Format:



  - Special unit for sign extension – bit 31 always sign bit!

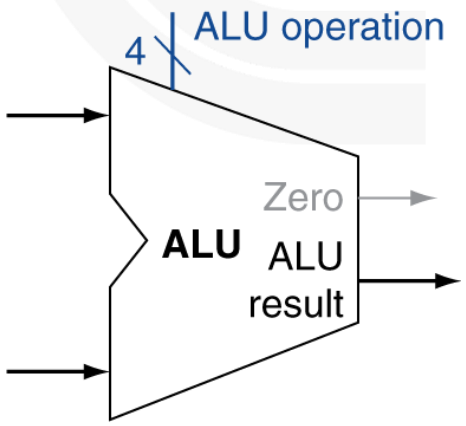
3. Execution

- **How do we know which operation to perform?**
  - R-Format: opcode: 7 bits, registers: 15 bits, 10 bits left
  - I-Format: opcode: 7 bits, reg/const: 22 bits, 3 bits left



- **Can be used to select ALU operation or other control**
  - Can encode 1000 different instructions with a single R-Format opcode!
- **R-Format (Register-Register) Instructions:**
  - `add`, `sub`, Shift Left/Right (`sll`/`srl`/`sra`), `and`/`or`/`xor`, Set less than (`slt`)
- **I-Format (Register-Immediate) Instructions:**
  - `addi`, `andi`, `ori`, `xori`, `slti`
  - No `subi` instruction. Just add a negative!

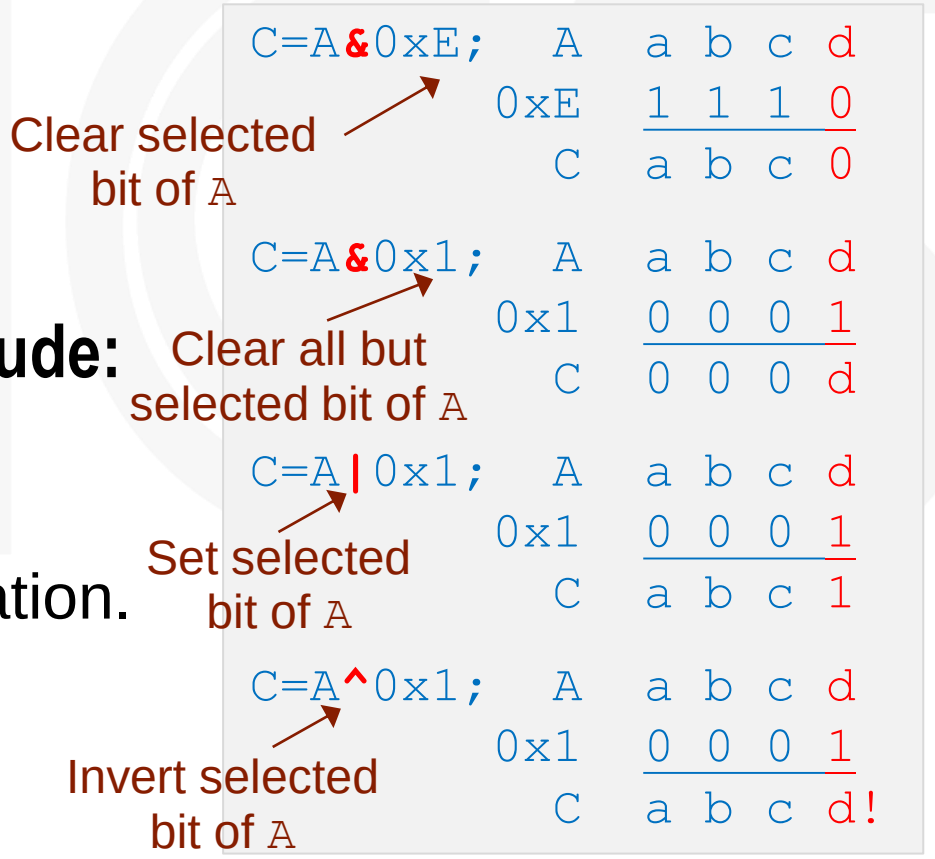
```
addi x1,x2,-0x123 → x1←x2-0x123
```



ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

What are bitwise operations used for?

- **We saw above that the ALU provides a variety of bitwise operations:**
  - `and`, `or`, `xor`, `andi`, `ori`, `xori`, `sll`, `srl`, `sra`, `slt`, `slti`
- **Similarly, C provides these operations:**
  - `&`(AND), `|`(OR), `^`(XOR), `~`(NOT), `<<`/`>>` (Shift)
- **But what are they good for?**
  - Use a "mask" to select bit(s) to be altered.
- **Common operations that one might perform, include:**
  - Set/reset bits on a microcontroller output port.
  - Testing status bits on input lines or in registers.
  - Set/reset status bits as the result of some operation.
  - Making comparison operations.
  - Quickly perform multiplication or division.



Variables and Memory Access

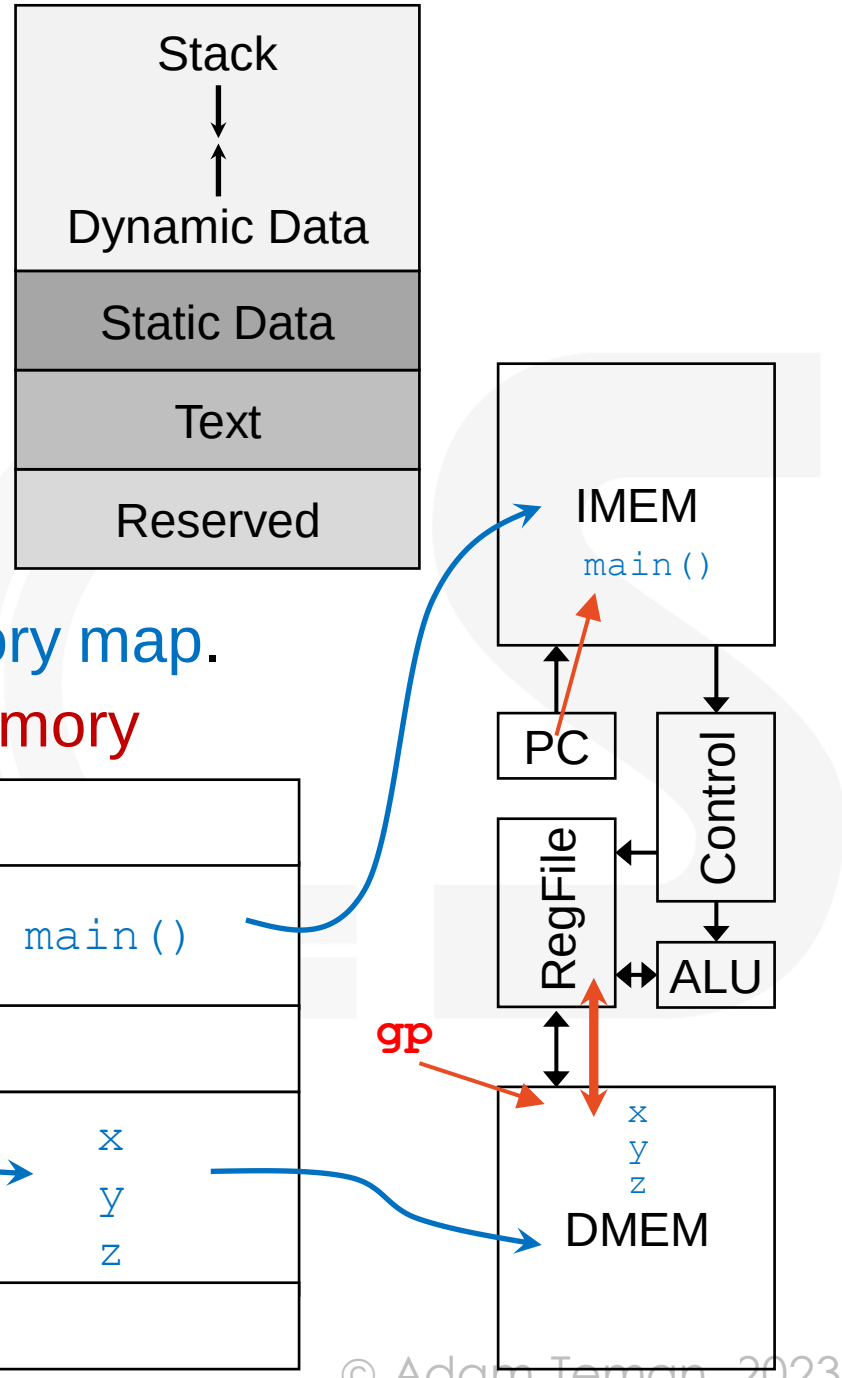
Emerging Nanoscaled Integrated Circuits and Systems Labs

The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

Program and Data Memory

- **Let's take a very simple program:**
  - The compiler translates the high-level language (C code) into machine language (binary).
  - Both instructions and data are mapped to the address space of the system according to the memory map.
  - Program code (text) is mapped to the instruction memory
  - Global variables (Static Data) are mapped to the data memory
  - Load/Store operations are applied to move the data in and out of the registers.

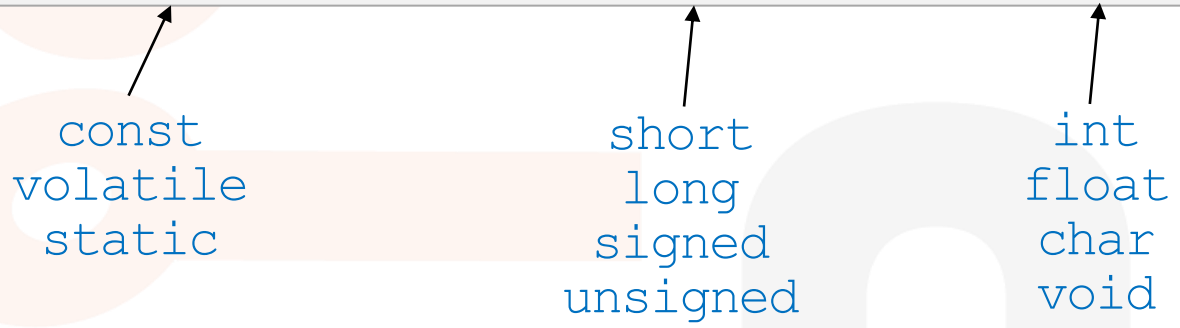
```
int main() {
    int x = 8;
    int y = 2;
    int z;
    z = x+y;
    return z;
}
```



C Variables

- **Variables in C are *declared*, *defined*, and *initialized*:**

```
type-qualifier(s) type-modifier data-type variable-name = initial-value;
```



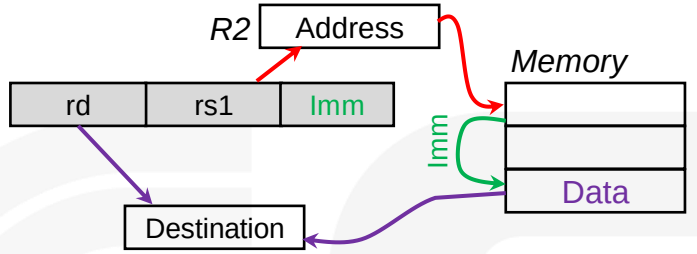

```
const unsigned char foo = 12;
long int bar;
```

- **Space for variables is allocated according to data type.**
  - Global and Static variables allocated in Static Data.
    - Static variables have scope of particular function.
  - Local ("automatic") variables allocated on the stack.
  - Compiler can also allocate local variables to registers.
  - **`volatile`** keyword ensures compiler will not remove.
  - Consts stored within read only section

C type	Bytes in RV32	Bytes in RV64
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`long long`	8	8
`void*`	4	8
`float`	4	4
`double`	8	8
`long double`	16	16

How are variables accessed in RISC-V?

- **Remember, RISC-V is a load-store architecture:**
  - There are no memory-to-memory operations.
  - All we need are commands to bring data from memory into a register and to write a result back into memory.



- **RISC-V only supports displacement addressing**
  - We achieve this with load and store instructions using the I/S-Formats.

  - `rs1` points to a register that holds the memory address, `Imm` defines the offset, and `rd`/`rs2` points to a register to load to or store from.
    - 12-bits provide an offset of up to 4096 bytes!
  - For example, word (32-bit) access uses the `lw` (load word) and `sw` (store word) commands.

```
lw x6, 123(x10) → x6=Mem[x10+123]
sw x6, 123(x10) → Mem[x10+123]=x6
```

How do we access an absolute address?

- **To access an absolute address, we need to load a 32-bit value into a register**
  - But the I-Format only provides room for a 12-bit value…
  - Instead let's use the 20-bit immediate in the U-Format

- **But how does this let us load a 32-bit constant into a register?**
  - Start with the load upper immediate (`lui`) instruction:
    - The immediate is loaded into bits `rd[31:11]`.
  - Now use `addi` to complete the 32-bit constant

```
lui  t0, 456→ t0=(456<<12)
addi t0, 789→ t0=(456<<12)+789
```

- **Another option is to create a PC relative address:**
  - Start with the add upper immediate to PC (`auipc`) instruction.
  - And complete with `addi`.

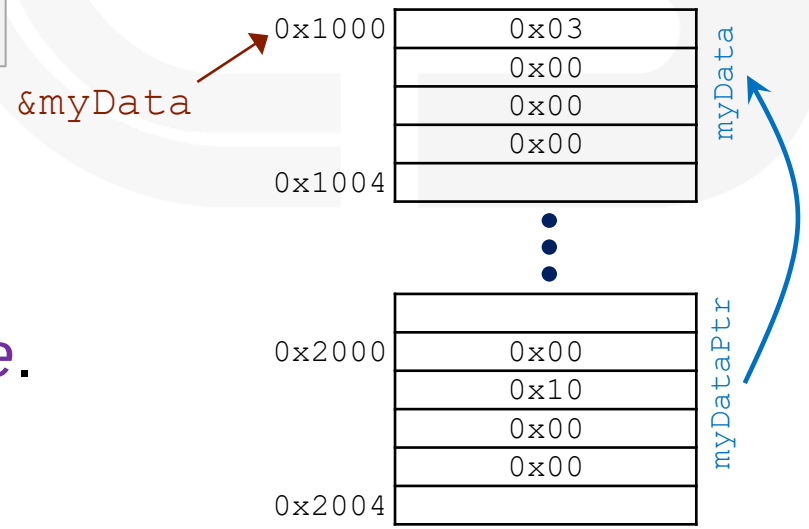
```
auipc t1, 456→ t0=PC+(456<<12)
addi  t1, 789→ t0=PC+(456<<12)+789
```

And what happens on the C side?

- **When we declare `int myData=0x03`, the compiler:**
  - Allocates 4 bytes of memory to hold the variable.
  - Places the value 3 (`0x03`) into those 32 bits.
  - Associates an address, such as `0x1000`, where the data will be stored.
  - Therefore, when we write `myData`, we are referring to data at address `0x1000`
- **We can store the address inside a special variable, called a pointer:**

```
int *myDataPtr = &myData;
```

  - `myDataPtr` is a variable of type "pointer to int" that stores the value `0x1000`.
    - In a 32-bit ISA (such as `RV32I`), a pointer is 4-bytes.
  - `&` is an operator that returns the *address of a variable*.
  - To read the declaration, go from right to left:
    - `myDataPtr`… is a pointer (`*`)… to an integer (`int`)



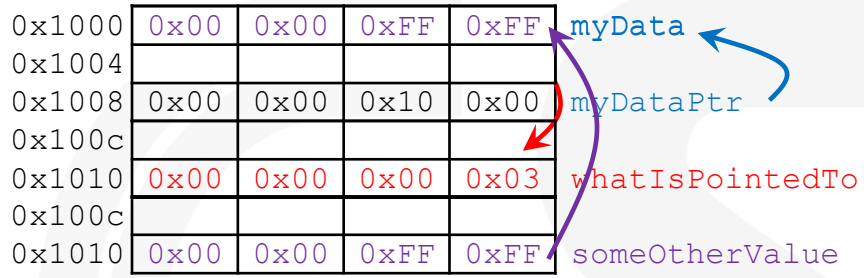
Pointer Arithmetic

- The **dereference operator (*) will return the value *pointed to* by a pointer**

```
int whatIsPointedTo = *myDataPtr;
```

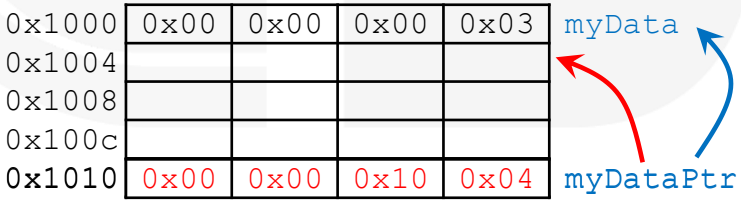
  - Assigning a value to a pointer will store the value at the address that is pointed to

```
int someOtherValue = 0xFFFF;
*myDataPtr = someOtherValue;
```

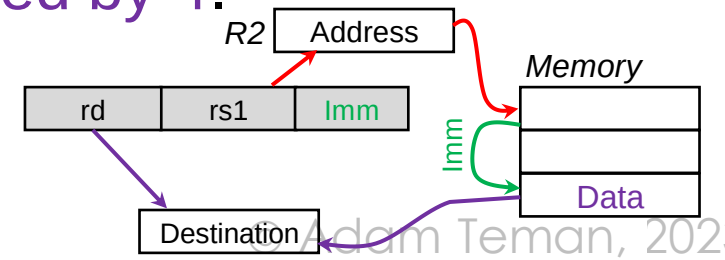


- **Adding a scalar value to a pointer is scaled by the datatype.**
  - Therefore, `myDataPtr++` is equivalent to

```
myDataPtr = myDataPtr + sizeof(*myDataPtr);
```



  - So, if `myDataPtr` is a pointer to int, then it is incremented by 4.
  - This is very useful for iterating over arrays.
  - RISC-V's displacement addressing makes this easy.



Arrays and Strings

- **An array is a set of items that have the same type and the same variable name.**
  - Array elements in C are stored in contiguous memory locations.

```
int  a[4]; // static array of 4 ints
char c[50]; // static array of 50 chars
```

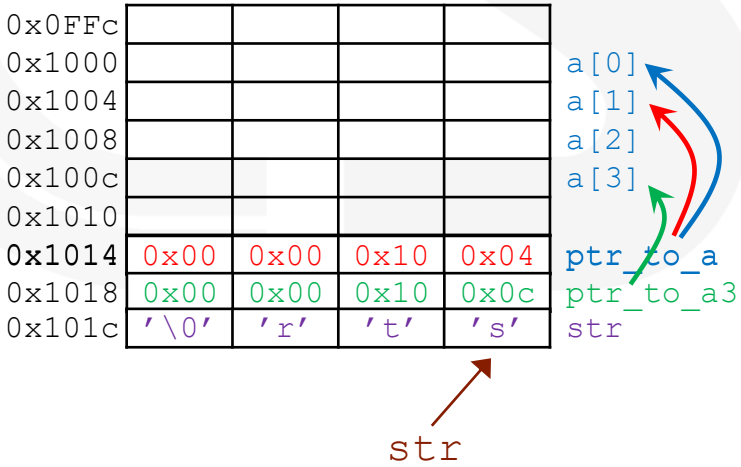
- **As such the address of the first element of an array is just a pointer to the array**

```
int *ptr_to_a = &a[0];   〓   int *ptr_to_a = a;
```

  - Incrementing the pointer will move it through the array

```
ptr_to_a++;   〓   ptr_to_a = &a[1];
```

```
int *ptr_to_a3 = &a[3];   〓   int *ptr_to_a3 = a+3;
```



- **A string is just an array of chars, ending with `null` (`\0`)**
  - And so, you can just declare it as a pointer with a literal.

```
char *str = "str";   〓   char str[4] = {'s','t','r','\0'};
```

Summary of Load/Store Execution

- C variables are stored in memory and loaded into registers for execution.
- Displacement addressing is used by placing a memory address (pointer) in a register and using an offset for array indexing.
- A 2R1W register file is used to access three operands for execution.

h is in t2

C Code:

```
A[12]=h+A[8];
```

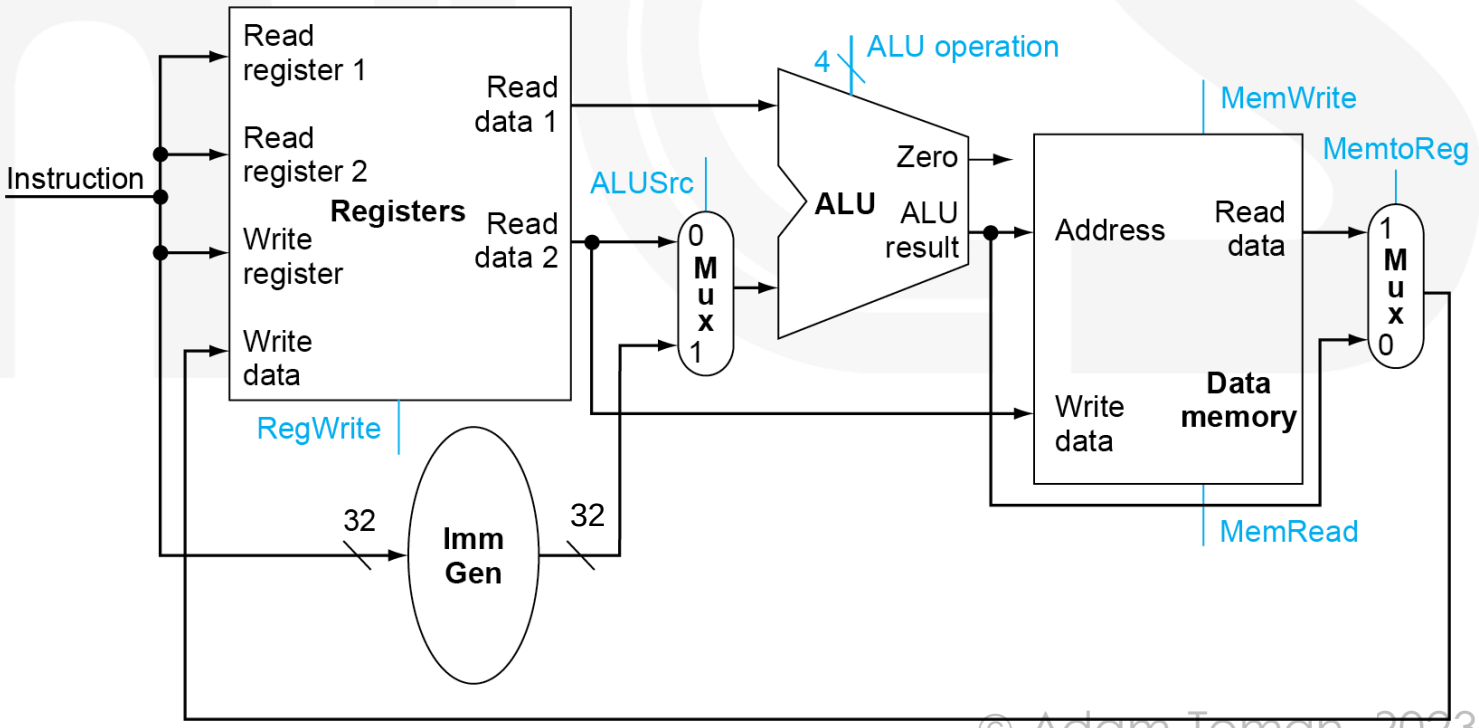
Base address of A is in t1

A[8] requires offset of 32

RV ASM:

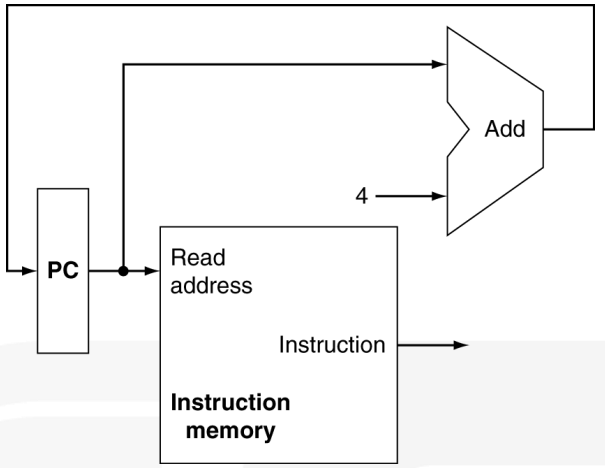
```
lw  t0, 32(t1)
add t0,t0,t2
sw  t0, 48(t1)
```

A[12] requires offset of 48



Control Flow

Control Flow and Conditionals



- **By default, the flow of a program will execute line-by-line**
  - At the ISA level, this means that the `PC` is incremented by 4 (one 32-bit instruction) every clock cycle.
- **Control flow constructs are a way of executing a segment of code if something is true or false.**

```
if (condition) {
    // run some code
}
```

- **Control flow at the ISA level is achieved using conditional branches**
  - Branch if greater than or equal Unsigned (`bgeu`)
  - Branch if less than/Unsigned (`blt/bltu`)
  - Branch if Equal/not Equal (`beq/bne`)
  - Branch if greater than or equal (`bge`)
- **Use the B-Format**

Enable $\pm 2^{12}$ branch (half word aligned)

C Code:

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

RV ASM:

```
        bne t0, t1, ELSE
        add t2, t3, t4
        beq x0, x0, EXIT
ELSE:   sub t2, t3, t4
EXIT:   ...
```

i j f g h

Loops

- C provides several means of implementing loops:

```
for (init; limit; incr)
   statement;
```

```
while (condition)
   statement;
```

```
do {
   statement;
} while  (condition)
```

- At the ISA level, these are all implemented with a conditional branch ("goto"):

C Code: (t6 t5 t7)

```
while (a[i]==k)
   i += 1;
```

RV ASM:

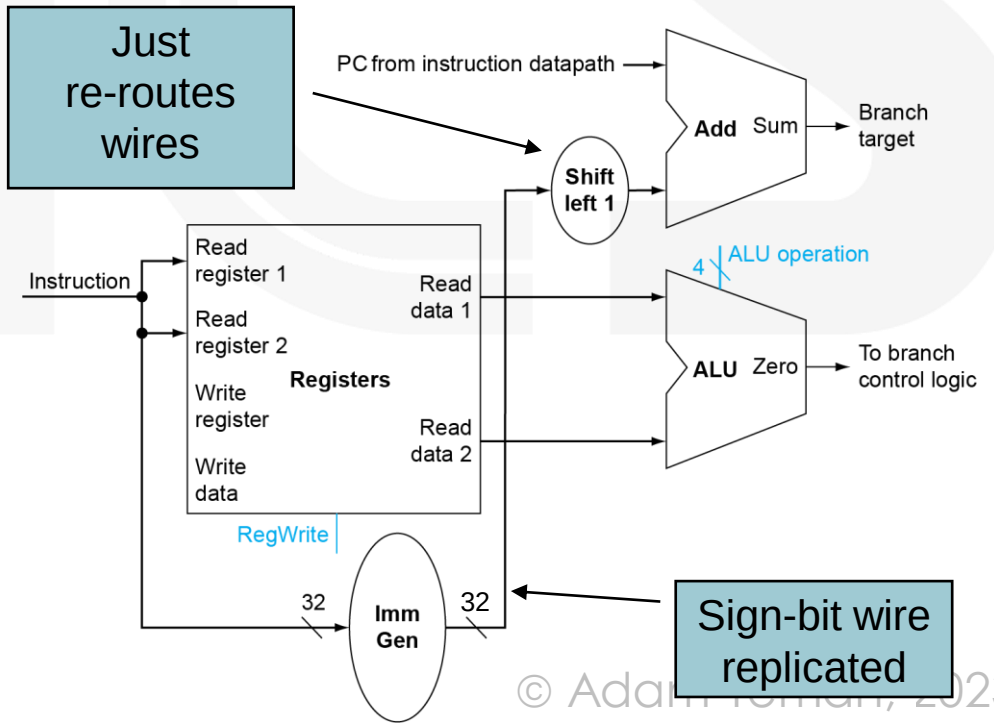
```
LOOP: slli t0, t5, 2
      add  t0, t0, t6
      lw   t1, 0(t0)
      bne  t1, t7, EXIT
      addi t5, t5, 1
      beq  x0, x0, LOOP
EXIT: ...
```

`t0` gets the address of `a[i]` and loads it into `t1`

`a[i]` is compared to `k`

`i` is incremented

unconditionally loop back



Procedure Calls

Emerging Nanoscaled Integrated Circuits and Systems Labs
The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

Functions

- **Functions (a.k.a., Procedures) are commonly used to:**
  - Make code modular, easier to read, easier to maintain.
  - Remove redundant copies of code.
- **Functions are often provided in two separate ways:**
  - Function Definition:
    Full description of the function, including the header and content
  - Function Declaration/Prototype:
    Abstraction of the function, only providing the header (interface)
- **This differentiation enables separating headers from code.**

```
func-type func-name( p1-type p1, p2-type p2, … ){
        //code for the function
}
```

```
func-type func-name( p1-type p1, p2-type p2, … );
```

Function Definition Example

```
int myfunc (int a, int b) {
   int c;
   c = a + b - 5;
   return (c);
}
```

Type of value to be returned to caller

Local variable

Arguments passed by caller

Return value

Function Prototype Examples

```
int main();
int myfunc( int a, int b );
char foo( char a, char b );
int * bar( float data );
void bar( int *ptr );
```

If no return value, use `void` type

Calling a Procedure

- **Calling a procedure is implemented in machine code by changing control flow to the address of the procedure in memory.**
  - This is achieved using an unconditional jump command.
  - Need to store the return address, a.k.a., "linking".
- **RISC-V provides two instructions for this:**
  - Jump and Link (`jal`) uses the J-Format to jump relative to the `PC`.
    - PC-relative displacement addressing.
    - Can jump $\pm 2^{20}$ bytes from the current address.
  - Jump and Link Register (`jalr`) uses the I-Format to jump to an absolute address.
    - Register displacement addressing.


```
jal ra,LABEL → PC←PC+Imm<<1, ra←PC+4
```


```
jalr ra,rs1,LABEL → PC←Imm(rs1), ra←PC+4
```

Passing arguments to a Function

- **Arguments are passed to a function primarily by:**
  - Using argument registers (`a0-a7`).
  - Pushing the arguments onto the stack.
- **The argument can contain either data or a memory address**
  - If data is passed, this is called "passing by value"
  - If an address is passed, this is called "passing by reference"
- **Passing by Value:**
  - The function header receives a regular C variable.
  - The function operates on a (read-only) copy of the variable.
  - Return values passed through registers (`a0-a1`) or the stack.
- **Passing by Reference:**
  - The function header receives a pointer.
  - The function can modify the actual variable.

Example – pass by value

```
int square (int x) {
  return (x * x);
}
void main {
  int k,n;
  n = 5;
  k = square(n);
  n = square(5);
}
```

Example – pass by reference

```
void sq (int x, int *y) {
  *y = x * x;
}
void main {
  int k, n;
  n = 5;
  sq(n, &k);
  sq(5, &n);
}
```

RISC-V Procedure Call

- **The RISC-V ABI defines:**
  - State registers, including the stack pointer (`sp`).
  - Saved registers, which a procedure will not change.
  - Temporary registers, which are "volatile", i.e., they may be changed by a called procedure.
  - Additional volatile registers for passing arguments and returning values.
- **The RISC-V Calling Convention requires that:**
  - The `sp` will *exit a function* with the value it had *when entering it*.
  - Registers `s0`-`s11` will *exit with the same value* as when entering.
  - The function will return to the address stored in `ra`.
- **In order to ensure this, a procedure always includes a prologue and epilogue.**

Reg.	ABI Name	Description	Saver
x0	zero	Hard-wired Zero	N/A
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	N/A
x4	tp	Thread Pointer	N/A
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Frame Pointer/ Saved Reg	Callee
x9	s1	Saved Register	Callee
x10-11	a0-1	Arguments/ Return Values	Caller
x12-17	a2-7	Arguments	Caller
x18-27	s2-11	Saved Registers	Callee
x28-31	t3-6	Temporaries	Caller

Prologue and Epilogue

- **To call a procedure, the caller will first:**
  - Place arguments in `a0`-`a7`
  - Store additional arguments and registers to save on the stack.
  - Call `jal` or `jalr` (placing `PC+4` in `ra`).
- **The callee then applies the prologue before the task:**
  - Allocate stack memory for variables and stored registers.
  - Store any saved register (`s0`-`s11`) it needs to overwrite.
  - Store `ra` on the stack, if a function call is made.
- **After finishing the task, the callee applies the epilogue:**
  - Reload registers that were saved on the stack (including `ra`).
  - Deallocate the stack: increment `sp` back to its original value.
  - Jump back to the return address using `jalr x0, ra`.

C Code:

```
int EXMPL (int g, int h)
{
    int f = SQR(g);
    f += h;
    return f;
}
```

RV ASM:

```
EXMPL:
    addi  sp,sp,-8
    sw    a1,8(sp)
    sw    ra,4(sp)
    addi  ra pc,0x4
    jalr  ra, SQR
    add   t0,x0,8(sp)
    add   a0,a0,t0
    lw    ra,4(sp)
    addi  sp,sp,8
    jalr  x0,0(ra)
```

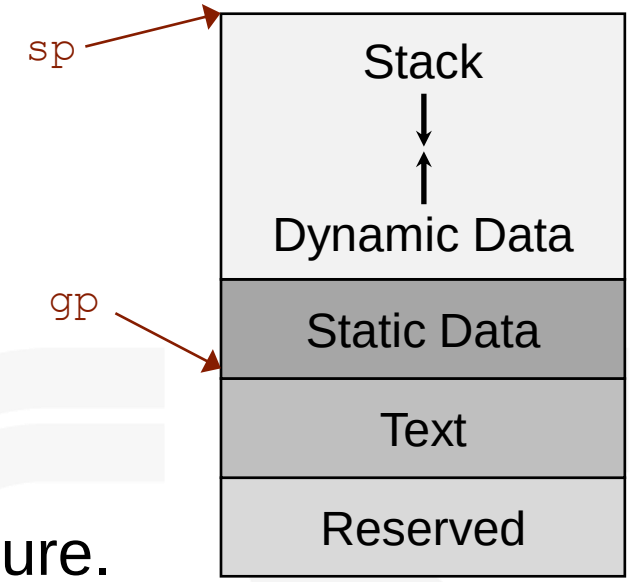
Prologue: store `h` and `ra` on stack

Function Call: `ra`←`PC+4` Jump&Link

calculate return value in `a0`

Epilogue: restore `ra`, `sp` and return

Variable Scope



Variables in C have "scope":

- **Local Scope**:
  - Variable defined within a function, including `main()`.
  - Local variables are stored on the stack frame of the procedure.
  - Upon returning, the memory is reclaimed and variables die.
- **Global Scope**:
  - Variables declared outside of all functions.
  - Global variables are stored in the Static Data region.
  - Global variables are accessible by all functions (using `gp`).
- **Static Scope**:
  - Variables declared as `static` are accessible by all instances of that function and automatically initialized to 0.
  - Static variables are also stored in the Static Data region.

global variable

static variable

local variable

```
int char a = 0;

int myfunc(int b); {
  static int c;
  c++;
  return (b+c);
}

void main {
  int k = 5;
  k = myfunc(10);
  a += myfunc(k);
}
```

RISC-V Features and Extensions

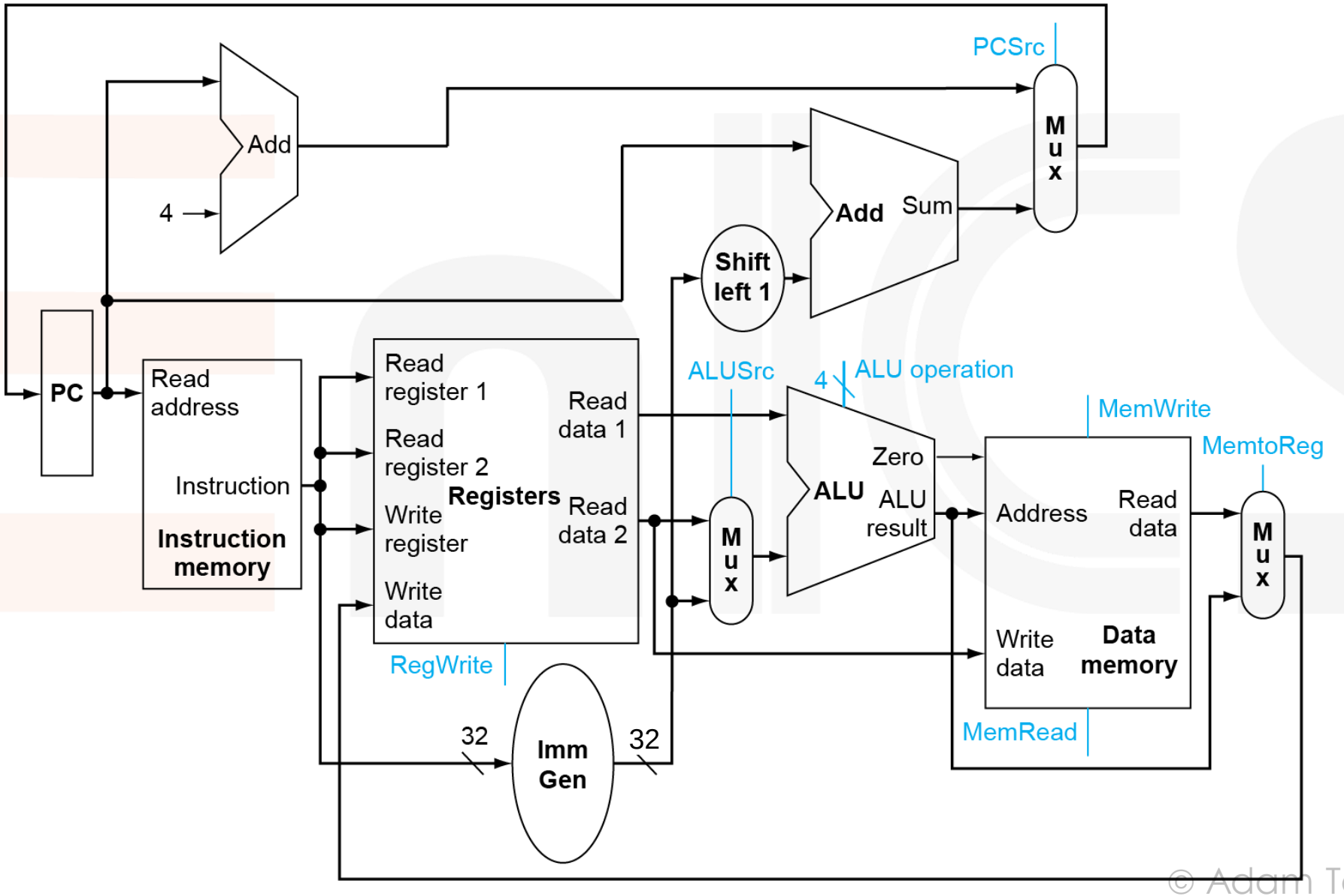
Emerging Nanoscaled Integrated Circuits and Systems Labs
The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

And a note about the ISA in general

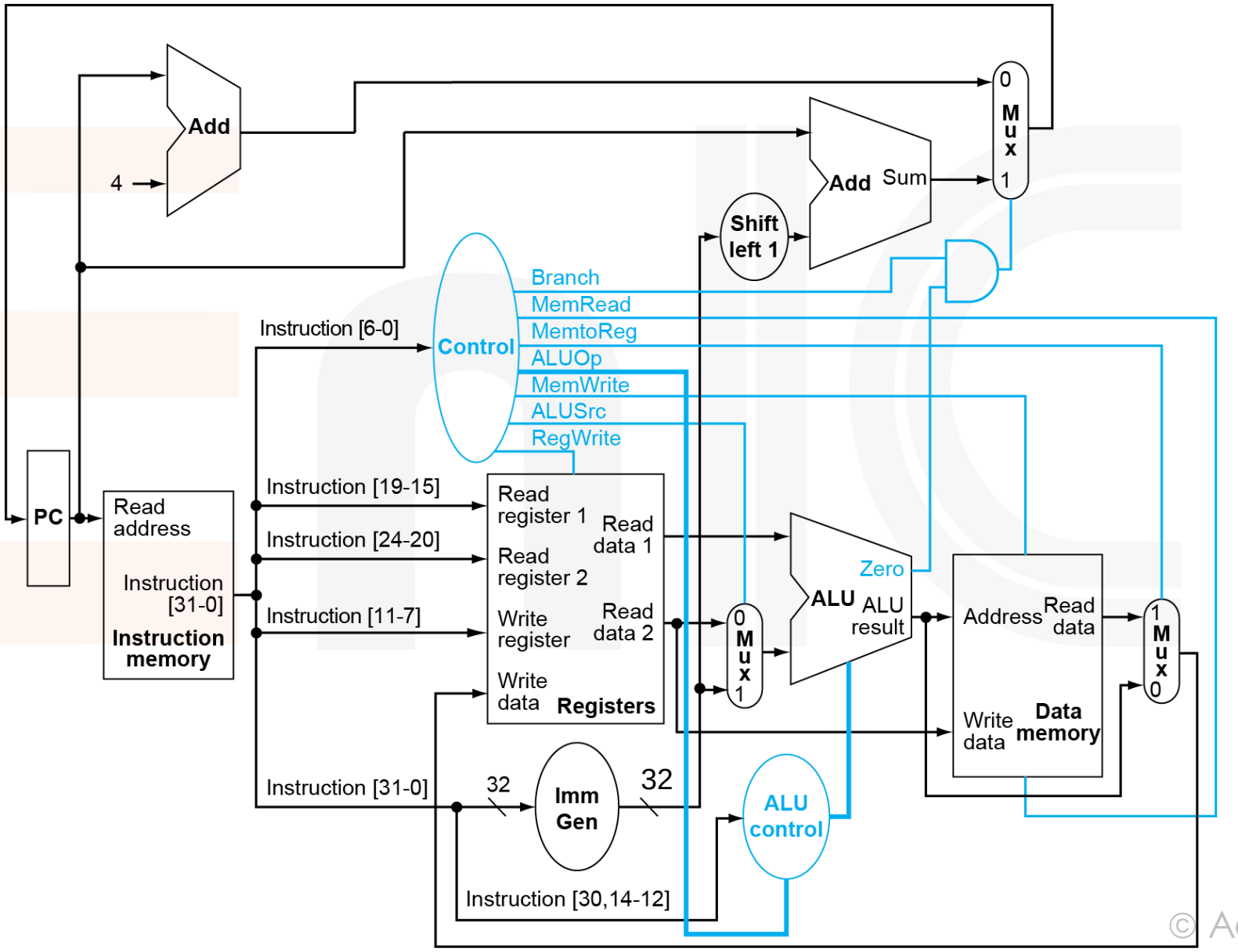
- **The RISC-V Base Integer ISA:**
  - Called: RV32I (32-bit), RV64I (64-bit), RV128I (128-bit)
  - *Must be present in any implementations*.
- **RISC-V is an *Extendable* architecture**
  - You can do all kinds of things to create additional instructions!
- **Standard instruction set extensions:**
  - **M**: integer multiply, divide, remainder
  - **A**: atomic memory operations
  - **F**: single-precision floating point
  - **D**: double-precision floating point
  - **C**: compressed 16-bit encoding for frequently used instructions
  - **E**: embedded – a smaller subset for small microcontrollers

RV32G means "All of the above"
→ Equivalent to **RV32IMAFD**

Full Base Architecture Datapath

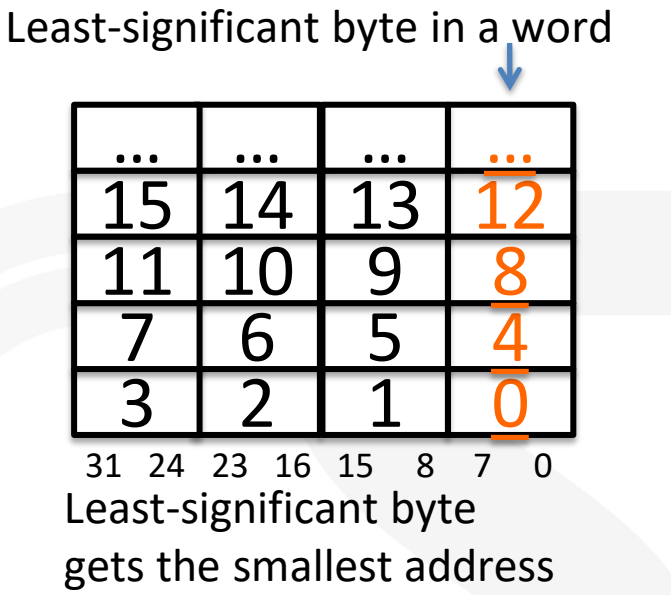


Base Architecture Datapath With Control



Additional Instruction Features

- **RISC-V is Little Endian**
  - Least-significant byte at least address of a word
  - c.f. Big Endian: most-significant byte at least address
- **RISC-V does not require words to be aligned in memory**
  - Unlike some other ISAs
- **RISC-V has no branch delay slots**
  - One of the big differences from MIPS.
- **No overflow checks on integer arithmetic.**
- **The 2-LSB bits are always 11.**
  - These bits are used for *compressed* instructions
- **All-zeros and All-ones instructions are illegal**



Overflow detection easily programmed

- For example, overflow detection of unsigned addition:

```
addi rd, rs, Immed-12
bltu rd, rs, OVERFLOW
     # if rd<rs then branch
```

Extensions

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 23.1: RISC-V base opcode map, inst[1:0]=11

- **Motivation**
  - RISC-V was developed to be *one ISA for all* (GP, embedded, accelerator).
  - Therefore, the lowest common denominator is the base (integer) architecture.
  - This is very small and simple and *must be present in all implementations*.
  - Any additional functionality is provided through *extensions*.
  - Extensions use the available instruction encoding bit space for identification.
- **Standard Extensions**
  - Generally useful and designed not to conflict with other standard extensions.
  - Examples are multiply/divide (M), atomic (A), floating point (FDQ).
- **Custom Extensions**
  - Highly specialized and may conflict with other standard extensions.
  - Can be of varying instruction length (e.g., 48b, 64b, etc.)

Compressed Instructions

- **RISC-V Instructions are 32-bit and *word-aligned*.**
- **However, the "C" Extension provides a set of 16-bit *half-word* aligned instructions.**
  - Each compressed instruction is *exactly equivalent* to some 32-bit instruction!
  - Since the most used instructions have a 16-bit equivalent, code size can be significantly reduced.
  - Smaller code size improves performance by more efficient caching.
- **With this extension, 32-bit and 16-bit instructions can be mixed freely.**
  - During decode, the 16-bit instructions are expanded into their 32-bit equivalent.
  - The 2 LSB bits of all 32-bit instructions are `11`. All others are compressed.
- **Many compressed instructions can only access certain registers (`x8` to `x15`)**
  - That way, for example, `add x8, x9, x10` can fit into 16-bits!
  - Other compressed instructions implicitly address certain registers according to the ABI, such as the stack pointer (`sp`) and return address (`ra`).
  - Some compressed instructions use 2-register addressing:

```
c.addw x8,x10 # x8=x8+x10 == add x8,x8,x10
```

Integer Multiplication

Two instructions are needed for multiplication

- First acquire high bits, then low bits

```
mulh rd-high,rs1,rs2
mul  rd-low, rs1,rs2
```

- **Multiplication (and Division) are part of the "M" extension.**
- **Notice what happens when multiplying two *n*-bit numbers:**

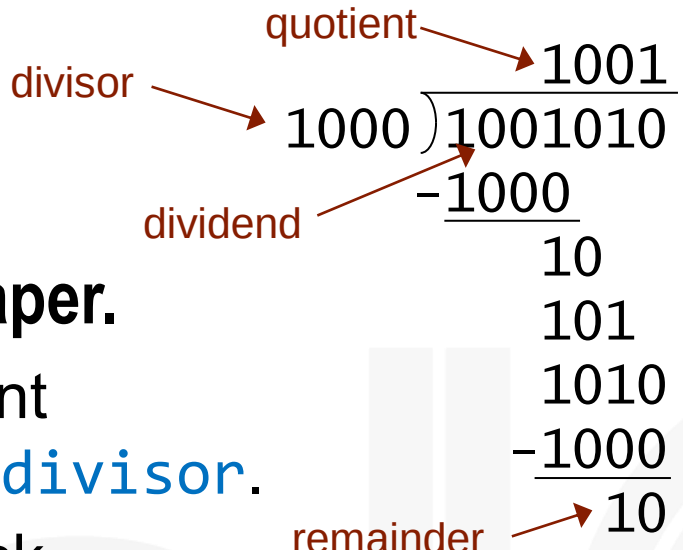
```
unsigned    1101 0101 = 213
unsigned  × 1011 1011 = 187
1001 1011 [1001 0111] = 39,831
```

```
signed      1101 0101 = -43
signed    × 1011 1011 = -69
0000 1011 [1001 0111] = 2,967
```

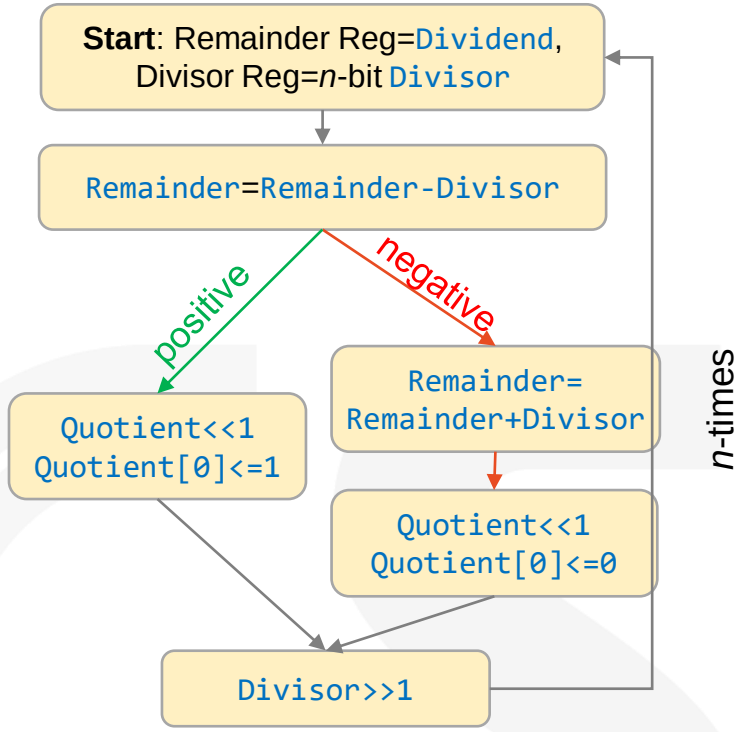
```
signed      1101 0101 = -43
unsigned  × 1011 1011 = 187
1110 0000 [1001 0111] = -8,041
```

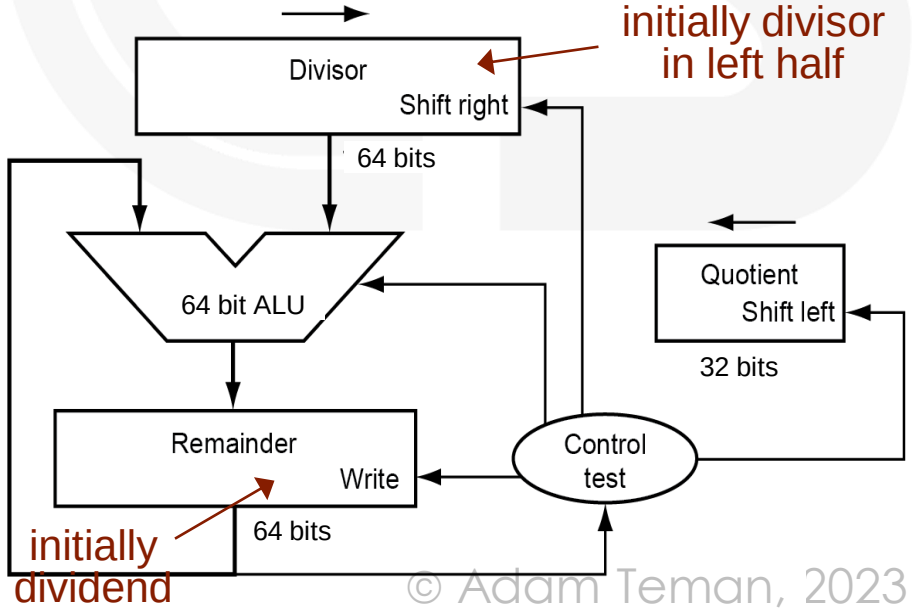
  - The result is $2n$ bits wide.
  - The bottom n bits are equal, regardless of signed/unsigned operands.
- **Therefore:**
  - `mul rd,rs1,rs2` – stores the n-lower bits in `rd`.
  - `mulh rd,rs1,rs2` – stores the n-higher bits in `rd` with signed operands.
  - `mulhu rd,rs1,rs2` – stores the n-higher bits in `rd` with unsigned operands.
  - `mulhsu rd,rs1,rs2` – has one signed and one unsigned operand.

Division



- **Divide like "long division" on paper.**
  - But we don't know if the current `remainder` is bigger than the `divisor`.
  - So we first subtract, then check.
    - If positive, `quotient` gets a `1`.
    - If negative, `quotient` gets a `0` and add back `divisor`.
    - In both cases, shift `divisor` left and `remainder` right.
    - After `n+1` steps, `quotient` and `remainder` are ready.
- **This is a long (n+1 cycle) process**
  - Faster algorithms exist, but are expensive.
- **The "M" extension has divide instructions:**
  - `div`/`divu`: Return `quotient` (signed/unsigned)
  - `rem`/`remu`: Return `remainder` (signed/unsigned)





Floating Point

- **Defined by IEEE Std 754-1985**
  - Single precision (32-bit) = `float`, Double precision (64-bit) = `double`

1 – negative
0 – non-negative

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- **Significand is always normalized:**
  - `1.0≤|significand|<2.0`
  - Binary floating point:

$$\pm 1.xxxxxxx_2 \times 2^{yyyy}$$

No need to represent leading '1'

`yyyy=exponent-Bias`
- Single bias: `127`
- Double bias: `1203`
- Exponent is unsigned

Like scientific notation:
- -2.34×10^{56} ← Normalized
- $+0.002 \times 10^{-4}$ ← Not Normalized
- $+987.02 \times 10^{9}$ ← Not Normalized

Floating Point Arithmetic

Addition:

Decimal Example

9.999×10^{1}
$+\ 1.610\times10^{-1}$

Align Binary Points

9.999×10^{1}
$+\ 0.016\times10^{1}$

Add Significands

10.015×10^{1}

Normalize Result

1.0015×10^{2}

Check overflow

Round and Renormalize

1.002×10^{2}

Binary Example

$1.000_2\times2^{-1}$
$+\ -1.110_2\times2^{-2}$

Align Binary Points

$1.000_2\times2^{-1}$
$+\ -0.111_2\times2^{-1}$

Add Significands

$0.001_2\times2^{-1}$

Normalize Result

$1.000_2\times2^{-4}$

Check overflow

Round and Renormalize

$1.000_2\times2^{-4}$

Multiplication:

Decimal Example

1.110×10^{10}
$\times\ 9.200\times10^{-5}$

Add Exponents

$10+(-5)=5$

Multiply Significands

1.11×9.20
$=\ 10.212\times10^{5}$

Normalize Result

1.0212×10^{6}

Check overflow

Round and Renormalize

1.021×10^{6}

Determine Sign

$+1.021\times10^{6}$

Binary Example

$1.000_2\times2^{-1}$
$\times\ -1.110_2\times2^{-2}$

Add Exponents

$(-1)+(-2)=-3$

Multiply Significands

$1.00_2\times1.11_2$
$=\ 1.110_2\times2^{-3}$

Normalize Result

$1.110_2\times2^{-3}$

Check overflow

Round and Renormalize

$1.110_2\times2^{-3}$

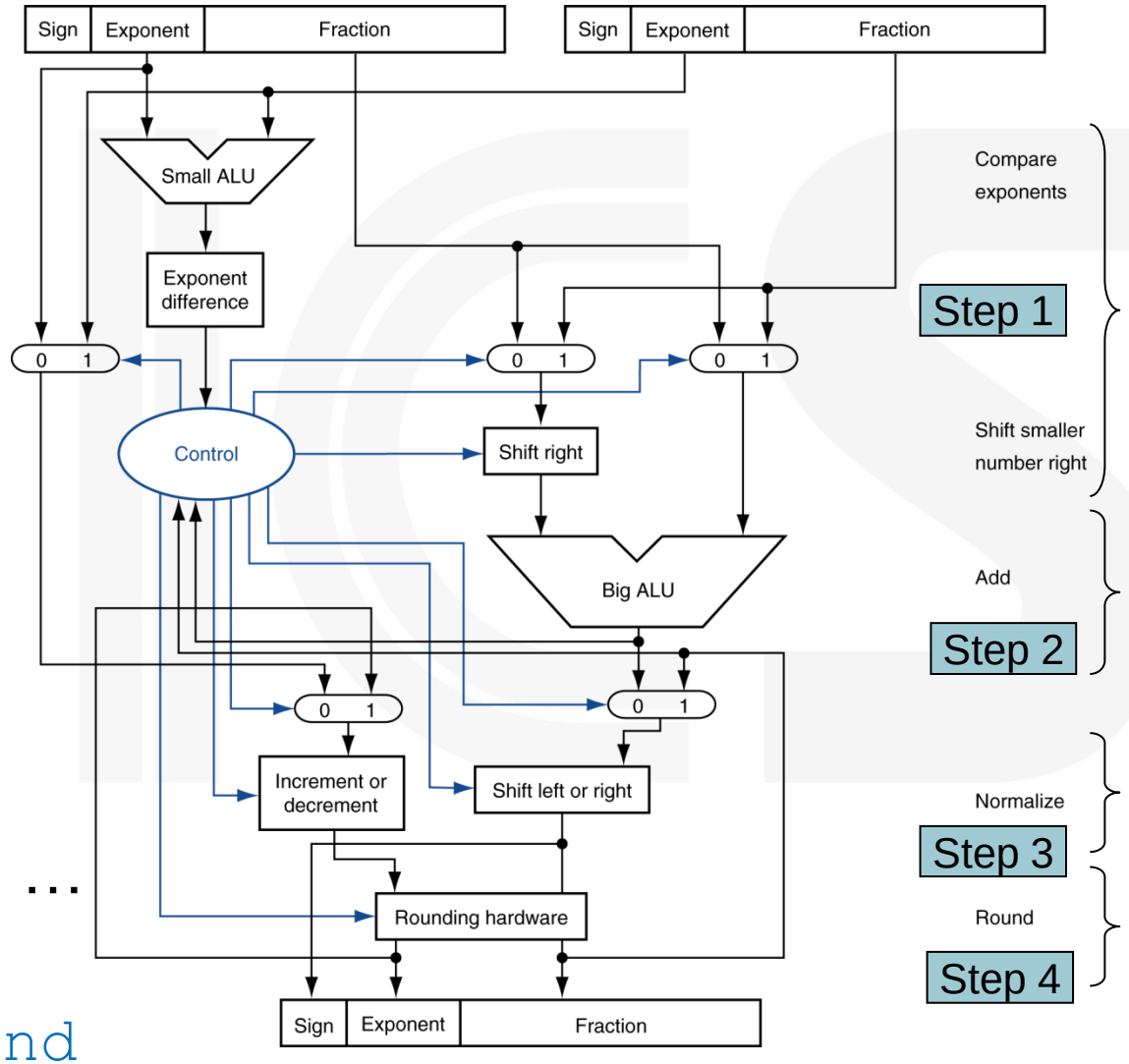
Determine Sign

$(+)\times(-)=(-)$

$-1.110_2\times2^{-3}$

Floating-Point Adder Hardware

- **Common FP Unit operations:**
  - Add/Sub, Mul/Div, Reciprocal, SQRT, FP↔Int Conversion
- **Much more complex than integer**
  - Operations take several cycles
  - Can be pipelined
- **RISC-V has special FP Registers**
  - Called `f0` to `f31`
  - FP load/store: `flw`, `fsw`
  - Arithmetic: `fadd.s`, `fmul.d`, `fsqrt.s`, …
  - Comparison: `feq.s`, `flt.d`, `fle.s`, …
  - Branch on FP condition true/false: `b.cond`

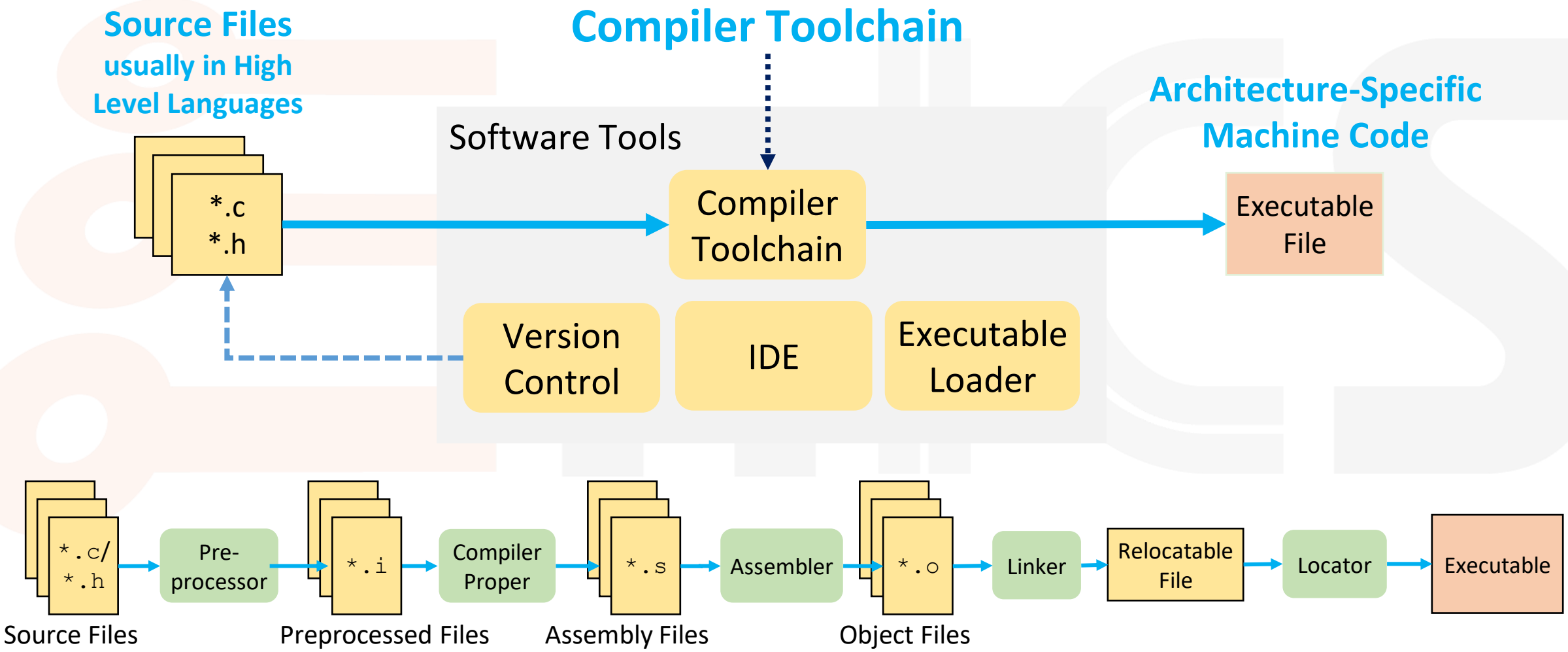


Additional Standard Extensions

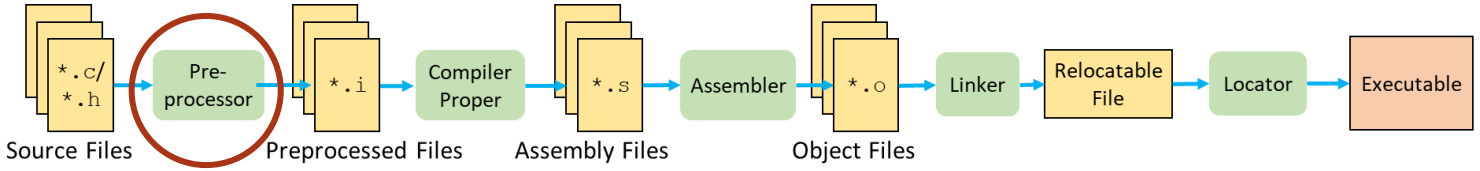
- Q - Quad-Precision Floating-Point
- L - Decimal Floating-Point
- B - Bit Manipulation
- J - Dynamically Translated Languages
- T - Transactional Memory
- P - Packed-SIMD Instructions
- V - Vector Operations
- N - User-Level Interrupts
- H - Hypervisor
- S - Supervisor-level Instructions

The Build Process (CALL)

Building a Software Project



Preprocessing



- **Before compilation, C code is sent through the preprocessor, in order to:**
  - Include external files (`#include`):

```
#include <stdio.h>
#include "mylibs.h"
```

  - Define constants, features and macros (`#define`):

```
#define PI 3.1415
#define CIRCLE_AREA(x) PI*x*x
```

  - Directives for Compilation Conditions (`#ifdef`, `#ifndef`, `#endif`, `#if`, `#else`):

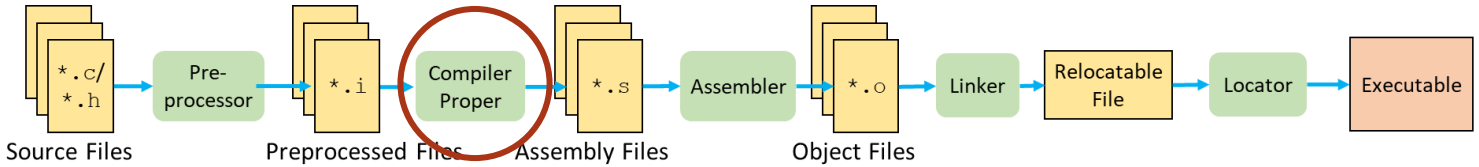
```
#define ADD_THE_FEATURE
#ifdef ADD_THE_FEATURE
    // void the_feature() { ... }
#endif
```

  - Passing instructions to the compiler (`#pragma`) and error reporting during compilation (`#error`).

source: freepik.com

source: safetymanagementgroup.com

Compilation



- **The compiler takes the preprocessed files (`.i`) and produces assembly files (`.s`):**
  - Readable text files according to the ABI.
  - Sectioning according to memory map.
- **Assembly code includes *pseudo-instructions* to make more readable, e.g.:**
  - `neg` ↔ `sub rd, x0, rs2`
  - `not` ↔ `xori rd, rs1,-1`
  - `nop` ↔ `addi x0, x0, 0`
  - `mv` ↔ `addi rd, rs1, 0`
  - `li` ↔ `addi rd, x0, Imm`
  - `ret` ↔ `jalr x0, x1, 0`.
  - `call` ↔ `lui/auipc + jalr`.
  - `j` ↔ `jal x0, LABEL`

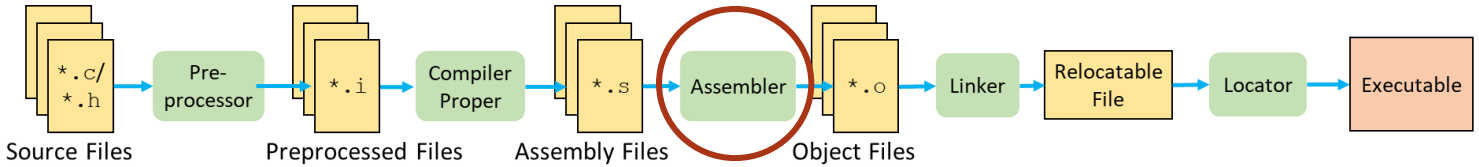
`hello.c`:

```
#include <stdio.h>
int main() {
   printf("Hello, %s\n",
                  "world");
   return 0;
}
```

`hello.S`:

```
.text
  .align 2
  .global main
main:
  addi sp,sp,-16
  sw   ra,12(sp)
  lui  a0,%hi(string1)
  addi a0,a0,%lo(string1)
  lui  a1,%hi(string2)
  addi a1,a1,%lo(string2)
  call printf
  lw   ra,12(sp)
  addi sp,sp,16
  li   a0,0
  ret
.section .rodata
.balign 4
string1:
  .string "Hello, %s!\n"
string2:
  .string "world"
```

Assembler



- **The Assembler translates Assembly code (`.S`) into binary object files (`.o`).**
- **The Assembler performs two passes over the code:**
  - First pass:
    Translate instructions/pseudo-instructions into binary.
    Remember the position of labels for forward references.
  - Second pass:
    Translate labels into immediates for branches and jumps.
- **But not all addresses can be calculated**
  - Only position-independent code (PIC) can be produced.
  - Absolute addresses calculated during linking/relocating.
  - Global and Static variables → Relocation Table
  - Labels from other files → Symbol Table
  - A standard format is ELF www.skyfree.org/linux/references/ELF_Format.pdf

```
   addi t2,x0,9
L1:slt  t1,x0,t2
   beq  t1,x0,L2
   addi t2,t2,-1
   j    L1
L2:
```

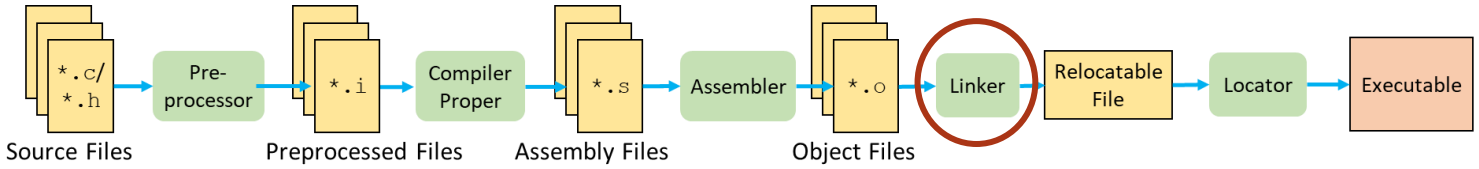
3 words forward (6 halfwords)

3 words **back** (6 halfwords)

hello.o:

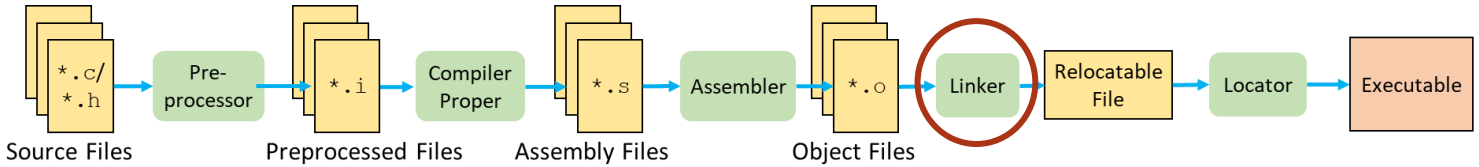
```
00000000 <main>:
0:  ff010113 addi  sp,sp,-16
4:  00112623 sw    ra,12(sp)
8:  00000537 lui   a0,0x0
c:  00050513 addi  a0,a0,0
10: 000005b7 lui   a1,0x0
14: 00058593 addi  a1,a1,0
18: 00000097 auipc ra,0x0
1c: 000000ef jalr  ra,0x0
20: 00c12083 lw    ra,12(sp)
24: 01010113 addi  sp,sp,16
28: 00000513 addi  a0,a0,0
2c: 00008067 jalr  ra
```

Linker



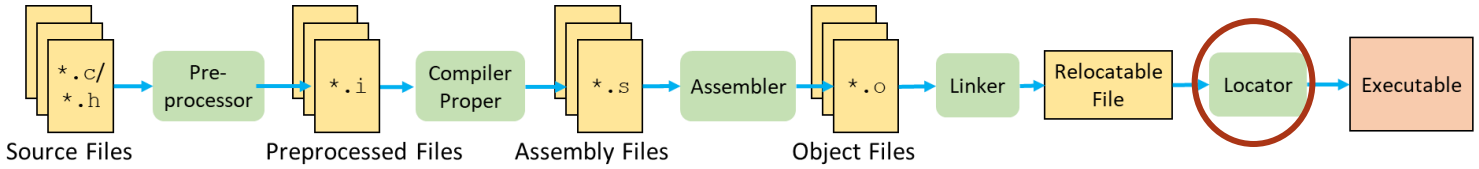
- **The linker combines several `.o` files into a single "relocatable" file.**
  - This includes two primary actions: Symbol Resolution and Relocation.
- **Symbol Resolution**
  - During assembly, some labels are "unresolved".
  - The linker looks for these labels in other files and copies them to the program.
- **Relocation**
  - During assembly, all programs start at address `0x0000`.
  - The linker merges all assembled files,
    and updates the instruction addresses (i.e., relocates the code).
- **The Linker creates a relocatable version of the program**
  - The program is complete, except no memory addresses assigned
  - The relocation table points to all labels that must be swapped with addresses.

Startup Code



- **During linking, special startup code is inserted into the program.**
- **Startup code for C programs usually does the following:**
  - Disables all interrupts
  - Copies initialized data from ROM to RAM
  - Zeroizes the unitialized data area
  - Allocates space for the stack
  - Initializes the stack pointer and global pointer
  - Enables interrupts
  - Calls `main()`
- **Startup code is usually provided as a file called `startup.asm` or `crt0.S`**

Locator



- **The final stage of the build process is the Locator.**
  - The Relocatable File contains the entire program but no memory addresses.
  - The linker script defines where different segments of memory should be stored.
  - The Locator replaces the placeholders (defined in the relocation table) with physical addresses, according to the linker script definitions.
- **The output is a binary memory image that can be loaded into the target ROM**
- **In embedded systems, the locator is often merged with the linker.**
  - In general purpose systems, relocation is performed during runtime by the loader.

`string1` is relocated to `20a10`

`string2` is relocated to `20a1c`

`printf` is relocated to `28800`

hello.out:

```
000101b0 <main>:
  101b0: ff010113   addi sp,sp,-16
  101b4: 00112623   sw   ra,12(sp)
  101b8: 00021537   lui  a0,0x21
  101bc: a1050513   addi a0,a0,-1520
  101c0: 000215b7   lui  a1,0x21
  101c4: a1c58593   addi a1,a1,-1508
  101c8: 288000ef   jal  ra,10450
  101cc: 00c12083   lw   ra,12(sp)
  101d0: 01010113   addi sp,sp,16
  101d4: 00000513   addi a0,0,0
  101d8: 00008067   jalr ra
```

Loader

- **While bare-metal embedded systems utilize startup code to run a program, higher-end computers running operating systems utilize a loader.**
- **A loader starts running an executable by:**
  - Reading the file's header to determine size of text and data segments.
  - Allocating address space for program, including `text`, `data` and `stack` segments.
  - Copying instructions + data from executable file into the new address space.
  - Relocating, Resolving Symbols and dynamically linking libraries.
  - Copying arguments (`argv`, `argc`) passed to the program onto the stack.
  - Initializes machine registers (`sp`, `gp`, etc.)
  - Jumping to start-up routine (`main()`)

References

- Patterson, Hennessy “Computer Organization and Design – The RISC-V Edition”
- Patterson, Waterman “The RISC-V Reader”
- Berkeley CS-61C, “Great Ideas in Computer Architecture”
- RISC-V Spec
- Harry H. Porter “RISC-V: An Overview of the ISA”
- Krste Asanovic, Hot Chips Tutorial on RISC-V, Aug. 2019
- USF C Tutorial, http://www.rc.usf.edu/tutorials/classes/tutorial/c_intro/
- Coursera, UC Boulder “Introduction to Embedded Systems”
- James Peckol, “Embedded Systems: A Contemporary Design Tool”