

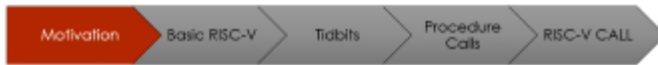
# Lecture 3: The RISC-V ISA

Advanced Digital VLSI Design I  
Bar-Ilan University, Course 83-614  
Semester B, 2021

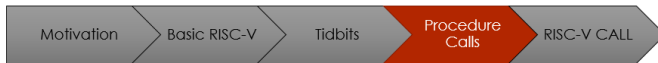
9 April 2021



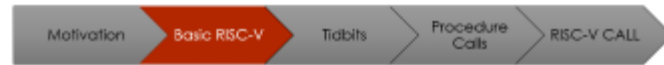
# Outline



## Motivation



## RISC-V Procedure Calls



## A Basic RISC-V Processor



## CALL in RISC-V



## Additional Tidbits

Motivation

Basic RISC-V

Tidbits

Procedure  
Calls

RISC-V CALL

# Motivation

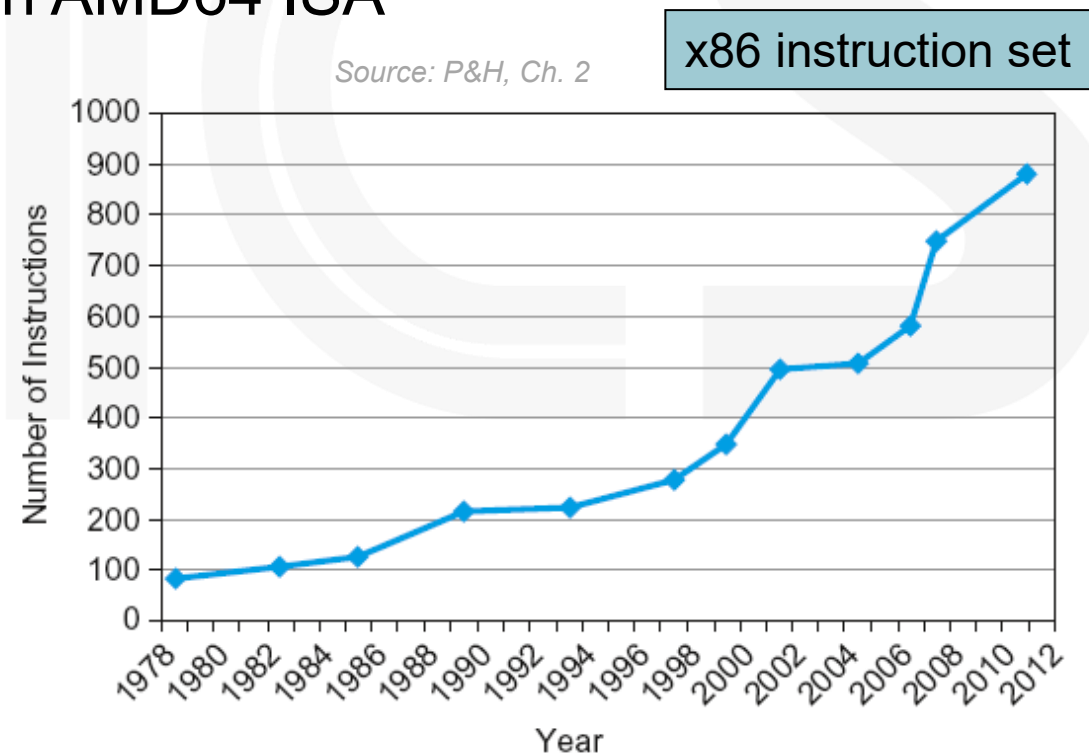


# Why **I**nstruction **S**et **A**rchitecture matters

- Why can't Intel sell mobile chips?
  - 99%+ of mobile phones/tablets based on ARM v7/v8 ISA (...now v9!)
- Why can't ARM partners sell servers?
  - 99%+ of laptops/desktops/servers based on AMD64 ISA  
(over 95%+ built by Intel)
- How can IBM still sell mainframes?
  - IBM 360, oldest surviving ISA (50+ years)

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

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



# Proprietary ISAs Die Out

- Proprietary ISA fortunes tied to business fortunes and whims

digital

VAX  
AlphaPowered

MIPS  
TECHNOLOGIES

Imagination  
WAVE  
COMPUTING

SPARC

Sun  
ORACLE



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

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

Source: K. Asanovic

Temam, 2021

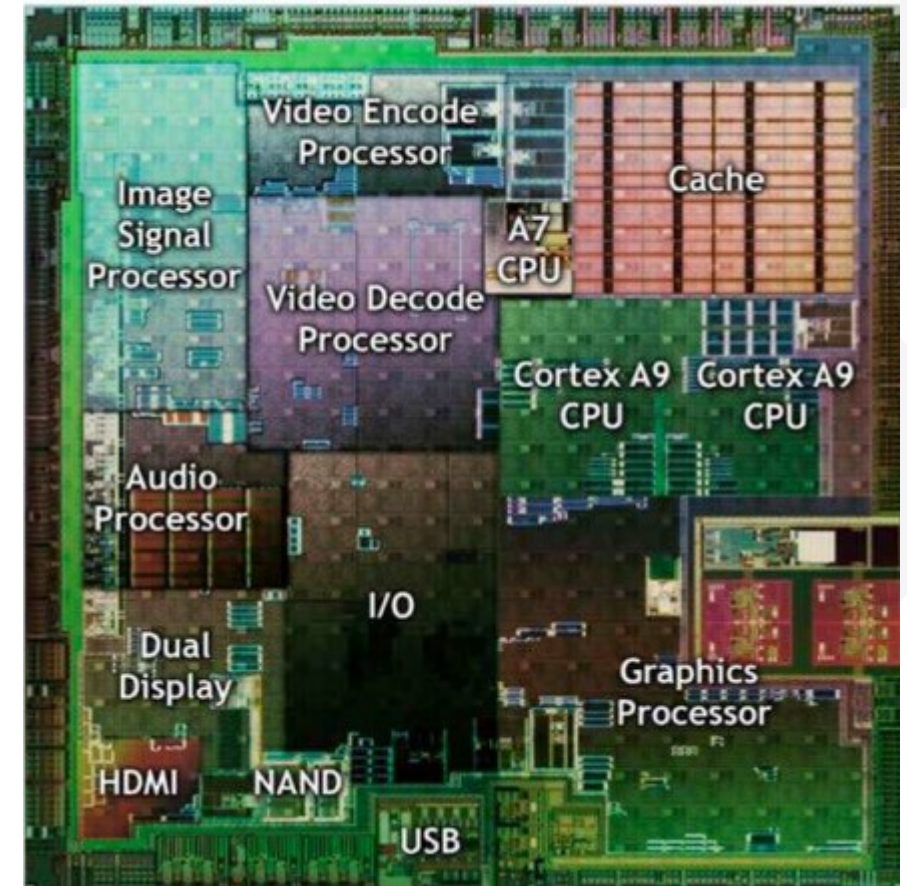
# The Need for a Single ISA

- 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

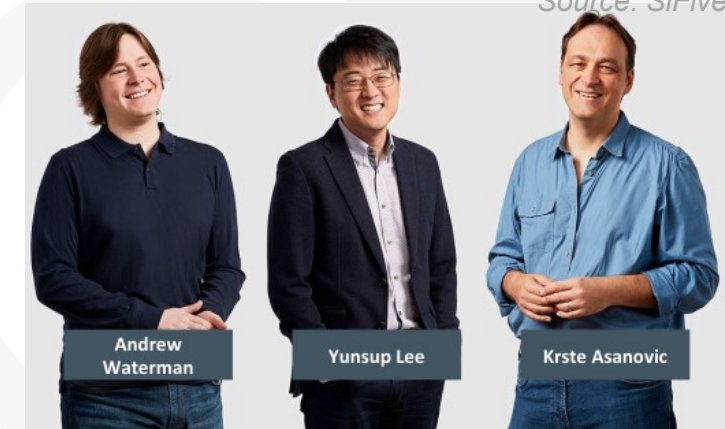


# The solution: RISC-V

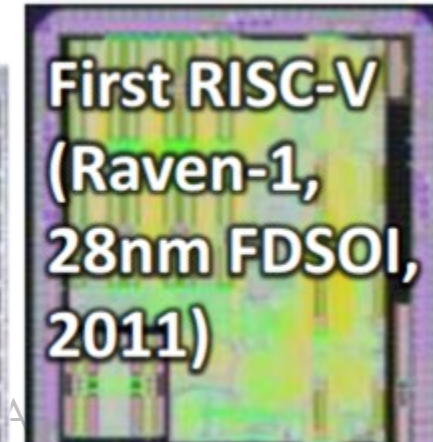


- Summer 2010: “3-month project” at UC-Berkeley
  - Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic
- May 2014: Frozen Base User Spec
- 2015: RISC-V Foundation Established
  - Led by Calista Redmond since 2019
  - Over 400 members (2020)
- RISC-V Project Goal:

*Become the industry-standard ISA for all computing devices*



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 microarchitecture 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



Motivation

Basic RISC-V

Tidbits

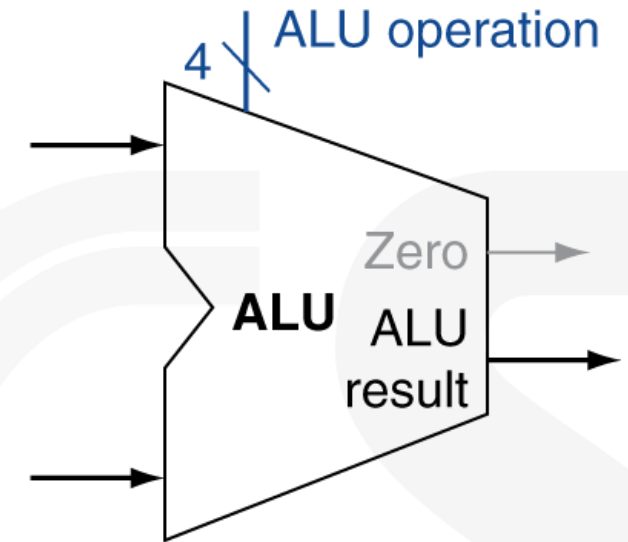
Procedure  
Calls

RISC-V CALL

# A Basic RISC-V Processor

# Basic Arithmetic Instructions

- Let's start with basic arithmetic instructions
  - Operate directly on registers
  - For example: `ADD x9, x20, x21` → `x9 ← x20 + x21`
  - Include: `ADD`, `SUB`, Shift Left/Right (`SLL/SRL/SRA`), `AND/OR/XOR`, Set less than (`SLT`)
- These operations have three components
  - Instruction Fetch
  - Register File Access
  - ALU Execution
- *Let us start building our datapath with these components*



- C code:

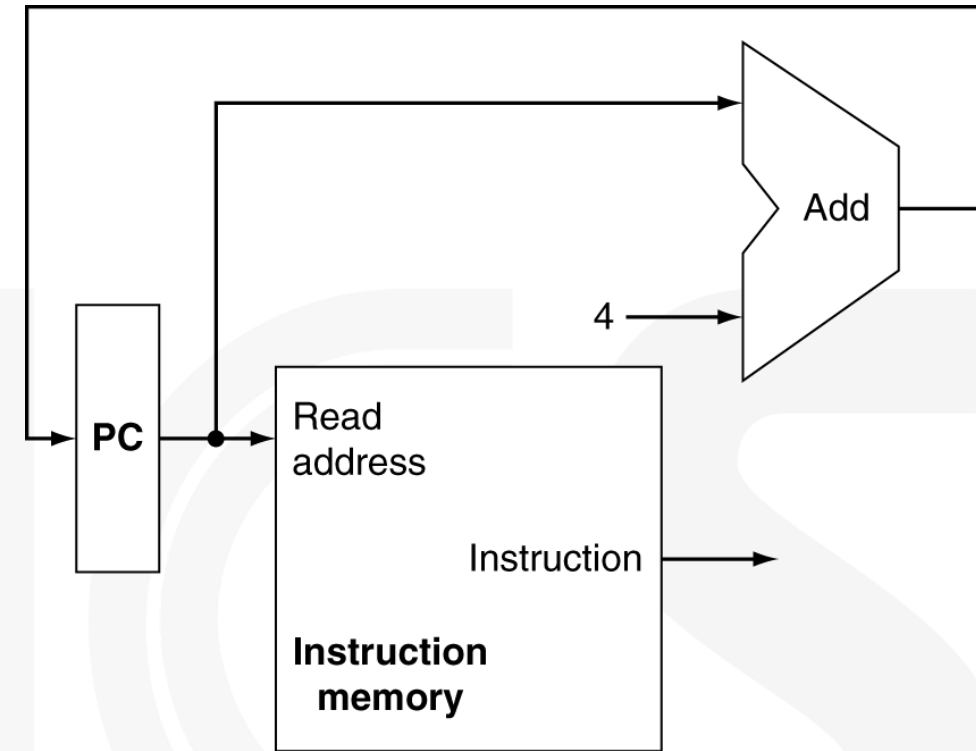
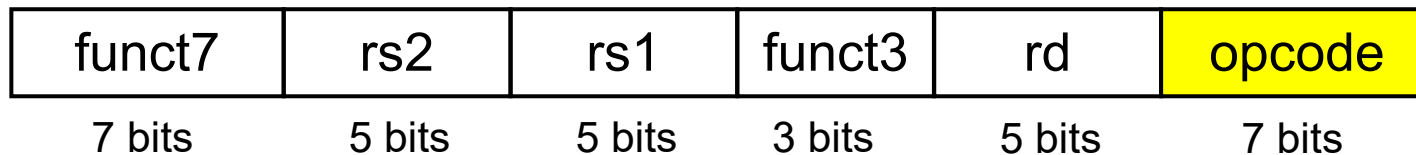
```
f = (g + h) - (i + j);
```

- Compiled RISC-V code:

```
ADD t0, g, h    // temp t0 = g + h
ADD t1, i, j    // temp t1 = i + j
SUB f, t0, t1   // f = t0 - t1
```

# 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
- Basic **register-register arithmetic** instructions use the **R-Format**



## 2. Register File Access

- 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)**

- Other “special” registers include:

- **x1**: Return address (**ra**)
- **x2**: Stack pointer (**sp**)      **x3**: Global pointer (**gp**)      **x8**: Frame pointer (**fp**)



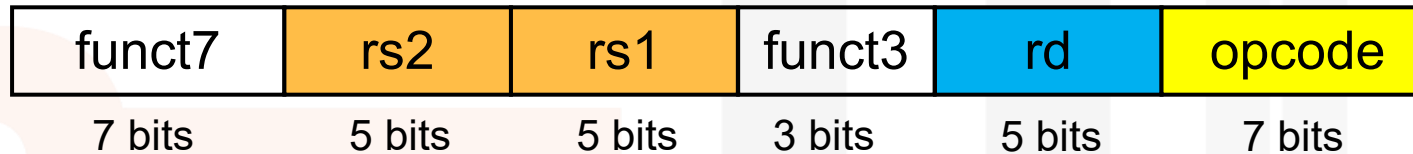
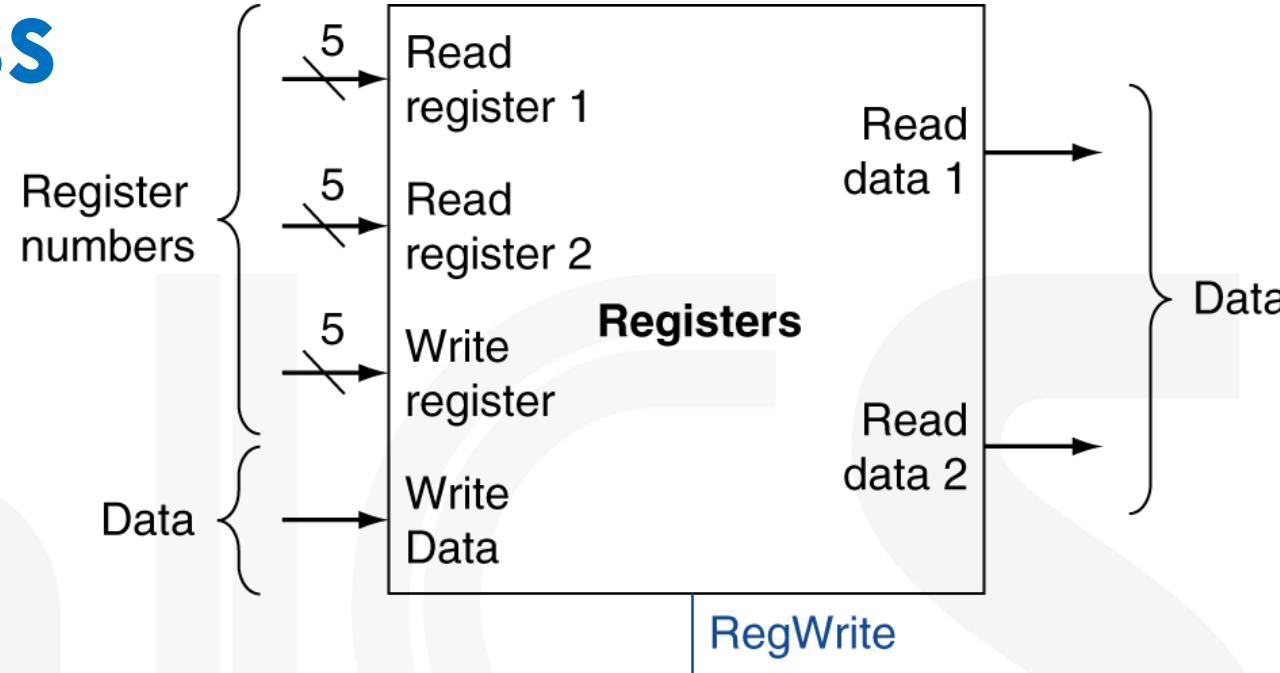
Source: Carl Goodwin

\*Registers are 32-bits wide for the RV32 ISA and 64 bits wide for the RV64 ISA

## 2. Register File Access

- **Register-Register Operations require:**

- Two source operands: **rs1**, **rs2**
- One destination operand: **rd**
- Register file requires **2R1W** access

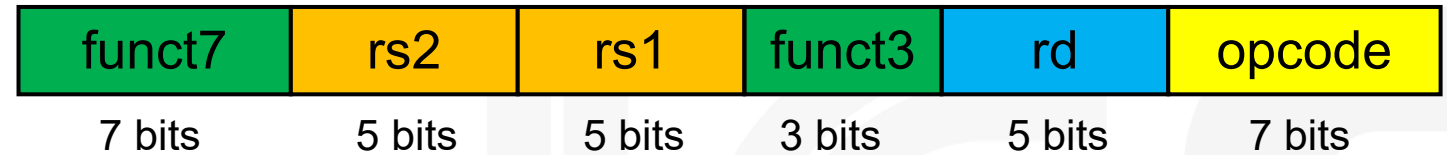




# 3. Execution

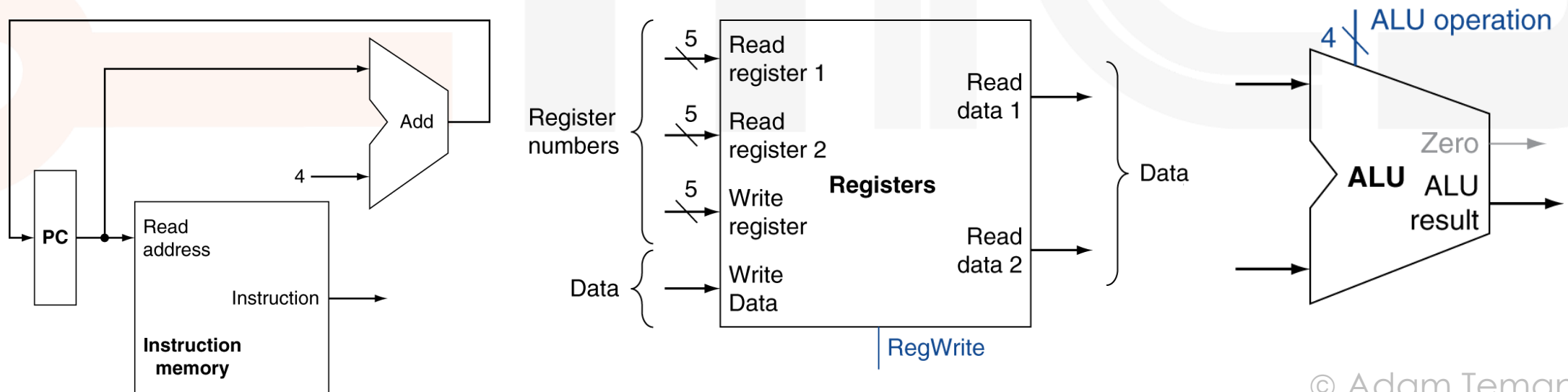
- **R-Format** Instruction

- 7 bits for opcode
- 15 bits for registers
- 10 bits left



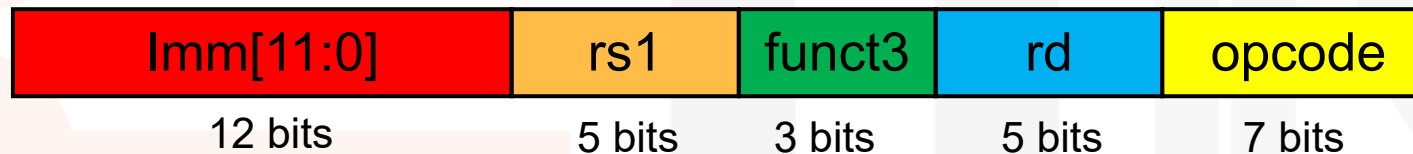
- Can be used to select ALU operation or other control

- Can encode 1000 different instructions with a single **R-Format** opcode!

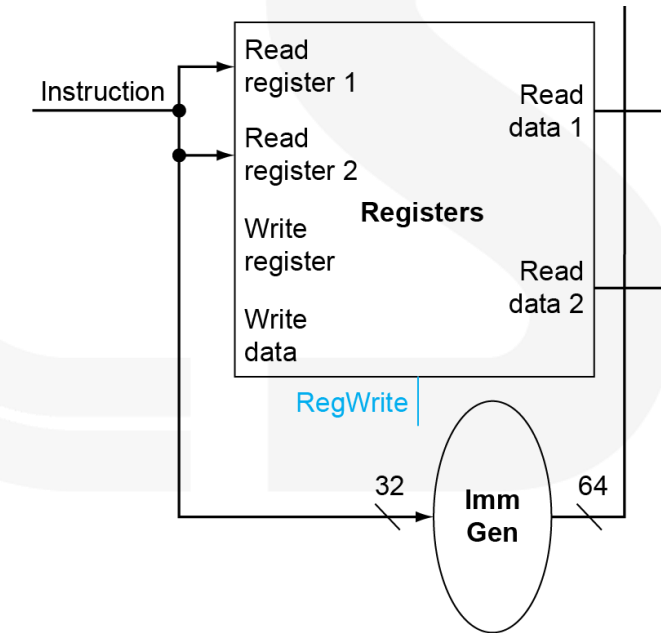


# Immediate Style Arithmetic

- What if we want to use a **constant** as one of the operands?
  - Loading the constant to a register would cost an entire instruction.
  - Can we save the instruction if we know the constant at compile time?
- Yes, use the spare bits for encoding constants
  - These are “**immediate**” instructions, given in the **I-Format**:



- For example: **ADDI x4,x6,123** → **x4=x6+123**
- Instructions include: **ADDI, ANDI, ORI, XORI, SLTI**
- Special unit for sign extension – **bit 31** always sign bit!
- No **SUBI** instruction. Just add a negative (**ADDI x4,x6,-123** → **x4=x6-123**)



# How about memory access?

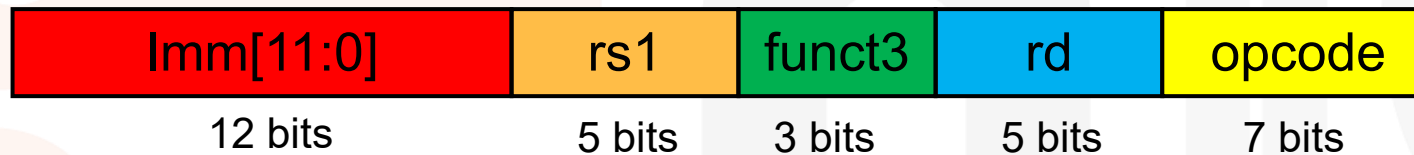
- RISC-V only supports **displacement addressing**

- For example:

LW x6, 123(x10) →  $x6 = \text{Mem}[x10 + 123]$

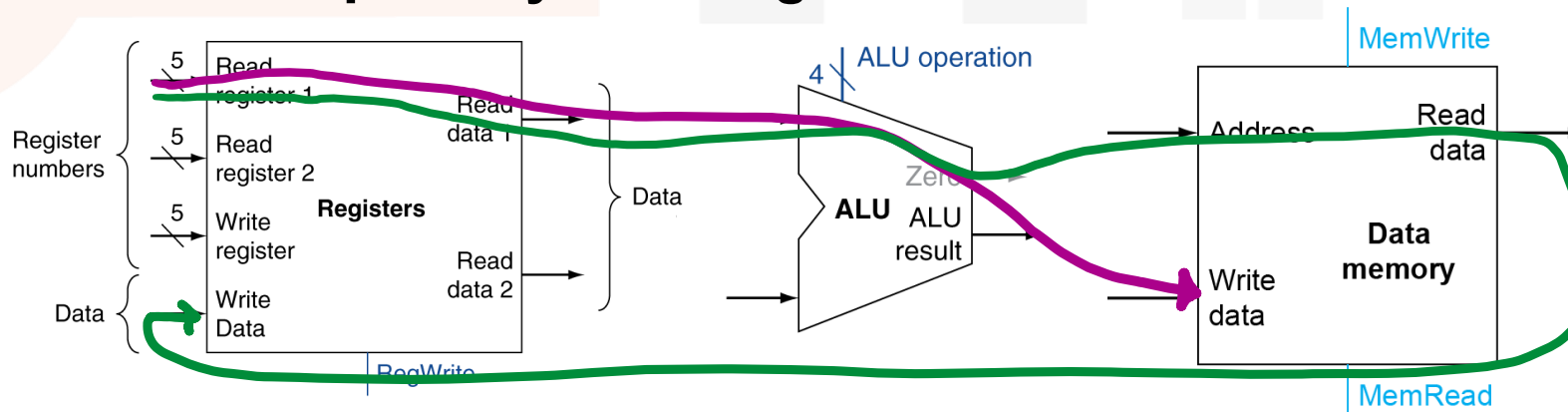
SW x6, 123(x10) →  $\text{Mem}[x10 + 123] = x6$

- This also uses the **I-Format**\*



So, we can address indexes as far as **4096 bytes** from the base address

- **Extend the datapath by sending the ALU result to the data memory:**



## C code:

```
A[12] = h + A[8];  
h in x21,  
base address of A in x22
```

## Compiled RV32I code:

- Index 8 requires offset of 32
- 4 bytes per word

```
LW    x9, 32(x22)  
ADD   x9, x21, x9  
SW    x9, 48(x22)
```

\*Stores use the similar S-Format just with rs2 instead of rd

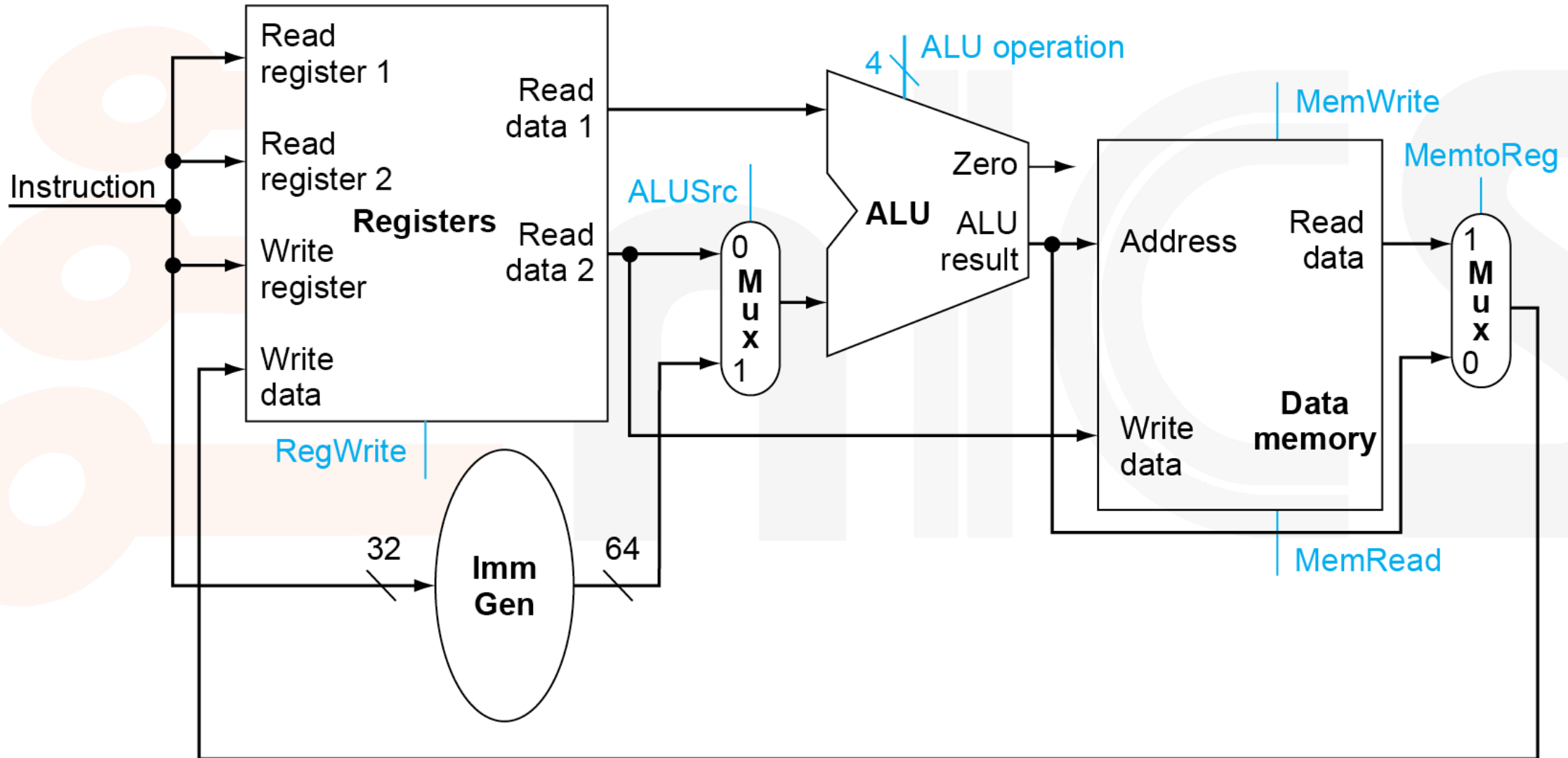
# How do we access an absolute address?

- 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...
- Introducing the **U-Format**:
  - Provides a **20-bit immediate** but only one register (**rd**).



- Use load upper immediate (**LUI**) together with **ADDI** to create **32-bit constant**
  - **LUI** x4, 123 → x4 = 123<<12
  - **ADDI** x4, x4, 456 → x4 = (123<<12)+456
- Or use add upper immediate to PC (**AUIPC**) to create **PC relative** address
  - **AUIPC** x4, 123 → x4 = PC+(123<<12)

# Summary: R-Type + Load/Store Datapath

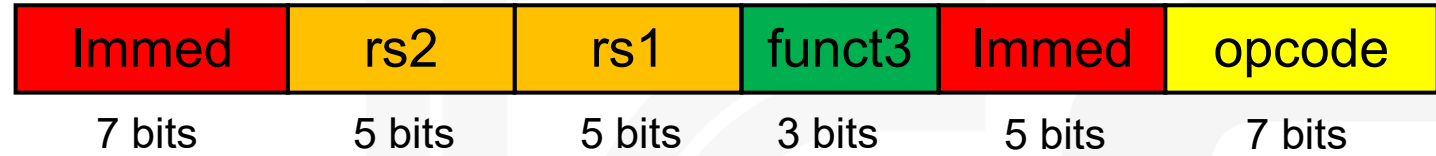




# Conditional Branch Operations

- Branch to label, depending on a certain condition

- Introduce the **B-Format**\*:



- e.g., **BEQ x6, x7, loop**

→ branch to label “loop” if  $x6 == x7$

- **rs1** and **rs2** for comparison

- Label encoded in **12-bit immediate** shifted left by 1 (for **2-byte** alignment)

- Allows branching to addresses up to **2048 bytes** away from **PC**.

- **Branching options include:**

- Branch if Equal/not Equal (**BEQ/BNE**)
- Branch if greater than or equal (**BGE**)
- Branch if greater than or equal Unsigned (**BGEU**)
- Branch if less than/Unsigned (**BLT/BLTU**)

\*Note that the B-Format is the same as the S-Format. The only difference is that the immediate is left shifted by 1-bit.

# Conditional Branch Operations

## IF Statements

- C code:

```
if (i==j) f = g+h;  
else f = g-h;  
• f, g, ... in x19, x20, ...
```

- Compiled RISC-V code:

```
        BNE x22, x23, Else  
        ADD x19, x20, x21  
        BEQ x0,x0,Exit //unconditional  
Else: SUB x19, x20, x21  
Exit: ...
```

## Loop Statements

- C code:

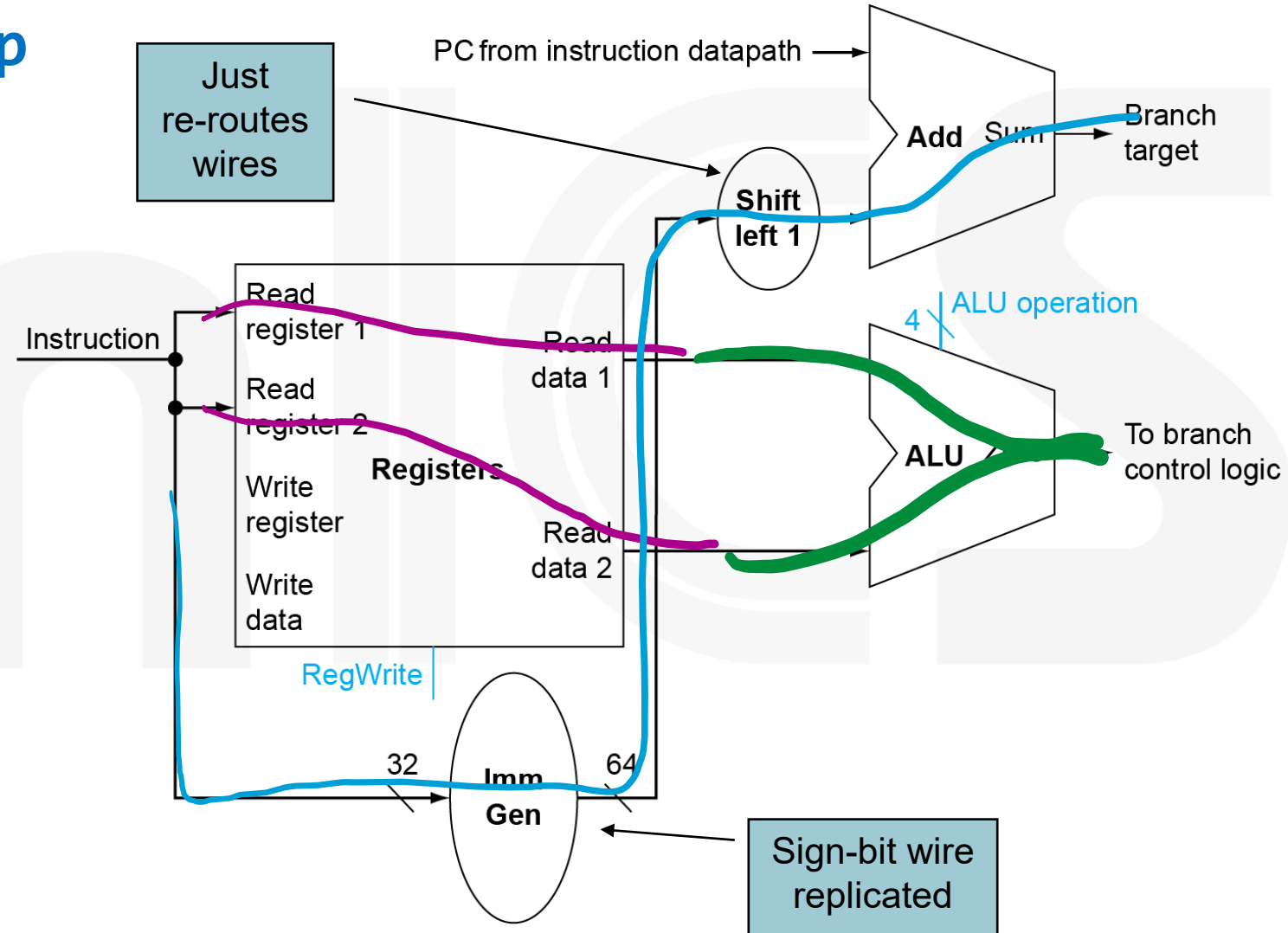
```
while (save[i] == k) i += 1;  
• i in x22, k in x24, address of save in x25
```

- Compiled RISC-V code:

```
Loop: SLLI x10, x22, 2  
      ADD  x10, x10, x25  
      LW   x9, 0(x10)  
      BNE  x9, x24, Exit  
      ADDI x22, x22, 1  
      BEQ  x0, x0, Loop  
Exit: ...
```

# Conditional Branch in the Datapath

- Example: **BEQ** x6, x7, loop
- Read register operands
- Compare operands
  - Use ALU to subtract
  - Check zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left by 1-bit (halfword displacement)
  - Add to PC value



# Unconditional Jump Commands

- Unconditional jumps are primarily used for **procedure calls**.
  - Need to store the return address, a.k.a. “**linking**”
  - Can be relative to the **PC** or to an absolute address stored in a register.

- **Jump relative to the PC:**

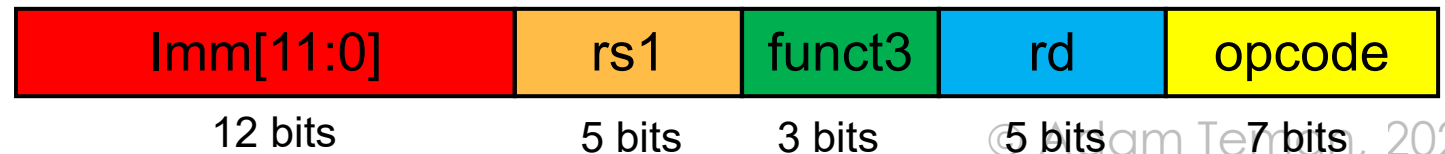
- Jump and Link (**JAL**)  
e.g., **JAL, x1, my\_procedure**
- Stores **PC+4** in **rd** and jumps to the **label**.
- Supported by the **J-Format**.



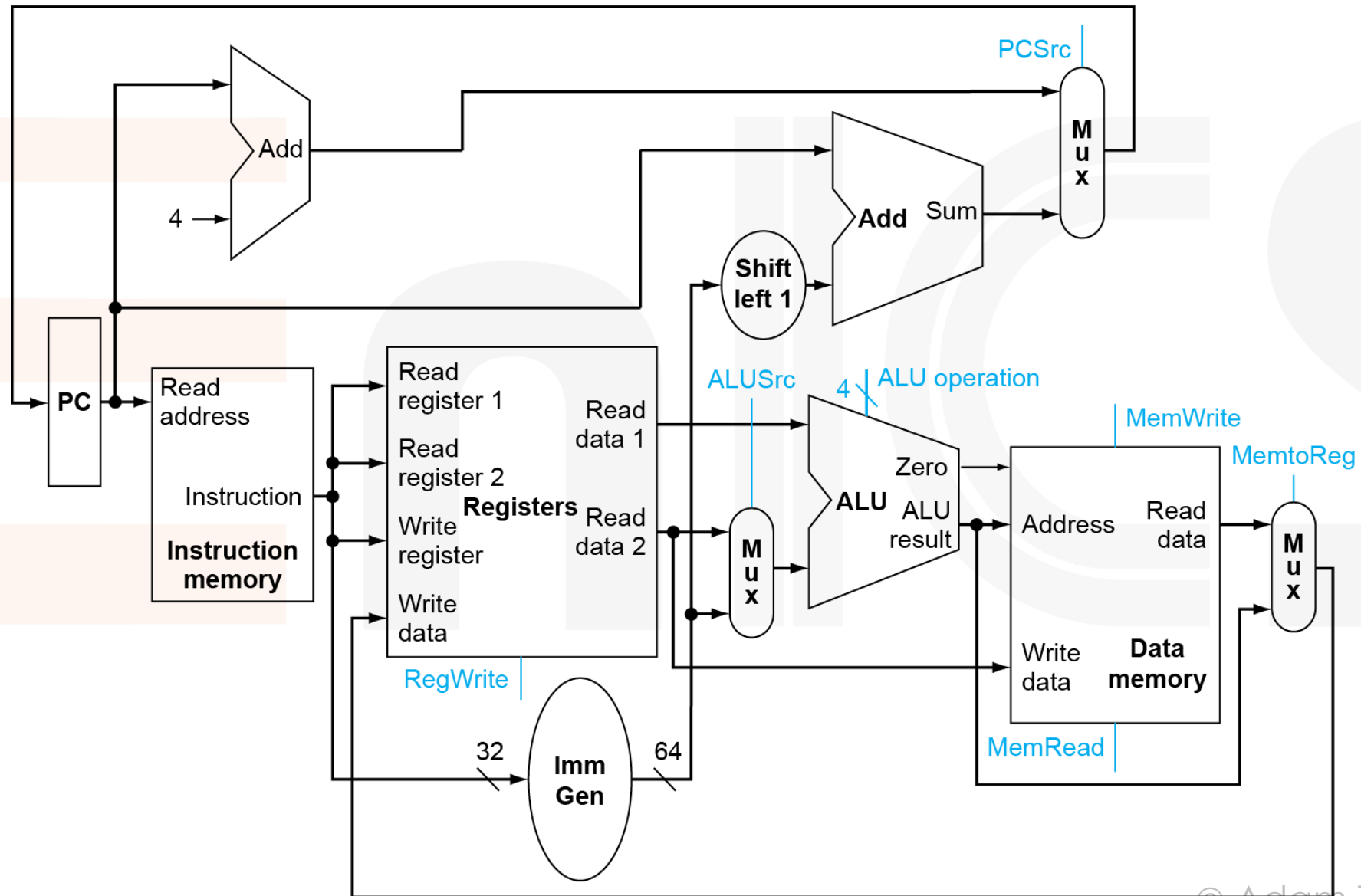
\*Note that the J-Format is the same as the U-Format. The only difference is that the immediate is left shifted by 1-bit not 20-bits.

- **Jump to an absolute address**

- Jump and Link Register (**JALR**)  
e.g., **JALR, x1, x6, 123**
- Stores **PC+4** in **rd** and jumps **imm(rs1)**.
- Uses the **I-Format**.



# Full Datapath





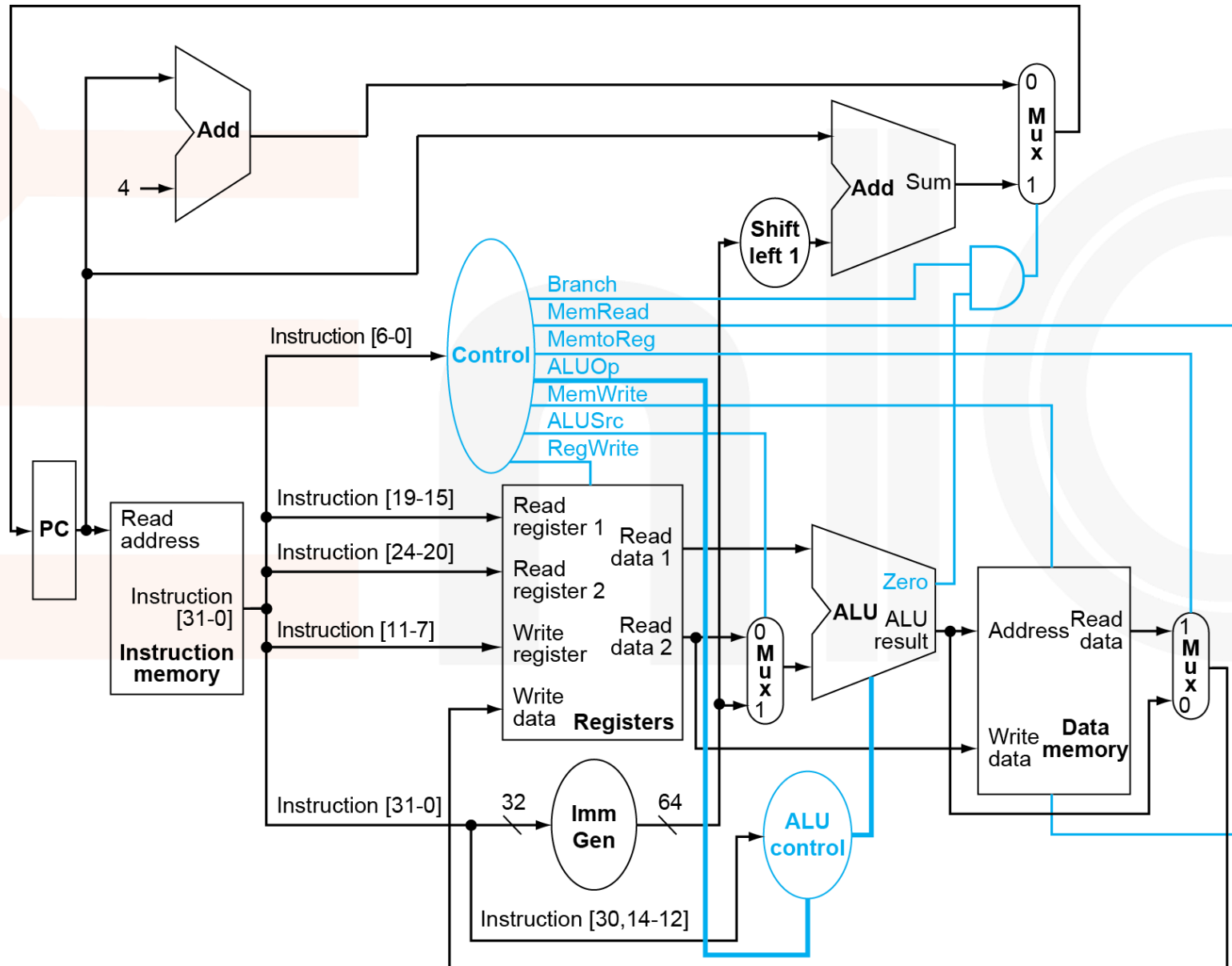
# Control Logic

- **ALU used for**
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode
- Assume **2-bit ALUOp** derived from **opcode**
  - Combinational logic derives ALU control

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
LD	00	load register	XXXXXX	add	0010
SD	00	store register	XXXXXX	add	0010
BEQ	01	branch on equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

# Datapath With Control



Motivation

Basic RISC-V

Tidbits

Procedure  
Calls

RISC-V CALL

# Additional Tidbits



# Review of Instruction Formats

## • R-Format

- For 3 register operations

OPR rd,rs1,rs2

## • I-Format

- For immediate instructions

OPI rd,rs1,Immed-12

## • S-Format (and B-Format)

- For store operations (and branches)

OPS rs1,rs2,Immed-12

## • U-Format (and J-Format)

- For long immediates (and Jumps)

OPU rd,Immed-20

rd – destination register  
rs1 – source register 1  
rs2 – source register 2  
Immed-12 – 12-bit immediate  
Immed-20 – 20-bit immediate

ADD x4,x6,x8 # x4=x6+x8

ADDI x4,x6,123 # x4=x6+123

LW x4,8(x6) # x4=Mem[8+x6]

SW x4,8(x6) # Mem[8+x6]=x4

BLT x4,x6,loop # if x4<x6 loop

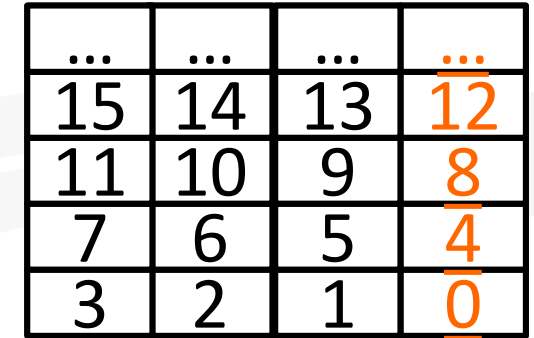
LUI x4,0x12AB7 # x4=value<<12

JAL x4,foo # jump&link

# 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-MSB** bits are always 11.
  - These bits are used for *compressed* instructions
- **All-zeros** and **All-ones** instructions are illegal

Least-significant byte in a word



...	...	...	...
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

31 24 23 16 15 8 7 0  
Least-significant byte  
gets the smallest address

## Overflow detection can easily be programmed

- For example, overflow detection of unsigned addition:  
`ADDI rd, rs, Immed-12`  
`BLTU rd, rs, OVERFLOW`  
*# if  $rd < rs$  then branch*



# Some Assembly Encodings

- Negate (**NEG**) is encoded as **SUB rd, x0, rs2**
- **NOT** is encoded as **XORI rd, rs1, -1 # -1=0xFFFFFFFF**
- No Operation (**NOP**) is usually encoded by **ADDI x0, x0, 0**
- Move (**MV**) is encoded by **ADDI rd, rs1, 0**
- Return (**RET**) is encoded as **JALR x0, x1, 0.**
- The **CALL** pseudo-instruction is used to call a faraway address, the offset will be made with **LUI/AUIPC** and **JALR**.
- Jump (**J**) is encoded as **JAL x0, LABEL**
- To obtain the current **PC**, use **AUIPC rd, x0**



Motivation

Basic RISC-V

Tidbits

Procedure  
Calls

RISC-V CALL

# RISC-V Procedure Calls

# Reminder Procedure Calls

- There are six stages in calling a function (a.k.a. **procedure**):
  1. Place the **arguments** where the function can access them
  2. **Acquire** necessary storage and **save the registers** that are needed
  3. Save **return address** and **Jump** to the function
  4. Perform the desired task
  5. **Return** from the function:
    - Place the **result values** where the calling function can access them
    - Restore any **registers**
    - **Release** any local storage resources
  6. Return control to the **point of origin**

Registers that need to be saved across a function call are called **saved registers**.

Registers that do not need to be saved are **temporary registers**.

The **stack pointer** is saved across function calls.

# Example of Procedure Call

- C code:

```
int leaf_example (int g, int h, int i, int j) {  
    int f;  
    f = (g + h) - (i + j);  
    return f;  
}
```

- Arguments g, ..., j in x10, ..., x13
- f in x20
- temporaries x5, x6
- Need to save x5, x6, x20 on stack

- RISC-V code (32-bit):

leaf\_example:

ADDI sp,sp,-12

Allocate Stack

SW x5,8(sp)

Save x5, x6, x20 on stack

SW x6,4(sp)

SW x20,0(sp)

ADD x5,x10,x11

x5 = g + h

ADD x6,x12,x1

x6 = i + j

SUB x20,x5,x6

f = x5 - x6

ADDI x10,x20,0

copy f to return register

LW x20,0(sp)

Restore x5, x6, x20 from stack

LW x6,4(sp)

LW x5,8(sp)

ADDI sp,sp,12

Deallocate Stack

JALR x0,0(x1)

Return to caller

High address

SP →

SP →

SP →

Low address

Contents of register x5

Contents of register x6

Contents of register x20

# Nested Procedures

- **Leaf Procedure**

- A procedure that does not call another procedure.

- **Nested (Non-Leaf) Procedures**

- Procedures that call other procedures
- But we will overwrite important data, such as the **return register**!

- **Therefore, we need to save the procedure state on the stack:**

- The procedure's **return address**
- Any **arguments** and **temporaries** needed after the call

- **After returning, restore the register state from the stack**

- C code:

```
int fact (int n) {  
    if (n < 1) return f;  
    else return n * fact(n - 1);  
}
```

- Argument  $n$  in  $x10$ , Result in  $x10$

- RISC-V code:

fact:

ADDI sp, sp, -8	Save return address and $n$ on stack
SD x1, 4(sp)	
SD x10, 0(sp)	
ADDI x5, x10, -1	$x5 = n - 1$
BGE x5, x0, L1	if $n \geq 1$ , go to L1
ADDI x10, x0, 1	Else, set return value to 1
ADDI sp, sp, 8	Pop stack, don't bother
JALR x0, 0(x1)	Return restoring values
L1: ADDI x10, x10, -1	$n = n - 1$
JAL x1, fact	call fact( $n-1$ )
ADDI x6, x10, 0	move result of fact( $n - 1$ ) to $x6$
LD x10, 0(sp)	Restore caller's $n$
LD x1, 4(sp)	Restore caller's return address
ADDI sp, sp, 8	Pop stack
MUL x10, x10, x6	return $n * \text{fact}(n-1)$
JALR x0, 0(x1)	return

# RISC-V Calling Convention

The RISC-V Calling Convention defines that when a procedure is called:

- The **caller** places the arguments in **argument registers a0-a7 (x10-x17)**.
  - If more space is needed (or larger arguments), the **stack** (or **memory**) is used.
  - These arguments are “passed by value”, so are local and disappear on return.
- The **caller** allocates stack space by moving the **stack pointer sp (x2)** down.
- The **callee** has to save the **saved registers s0-s11 (x8,x9,x18-x27)**
  - If the function doesn't use any of these registers then it doesn't have to save them.
  - The **callee** is allowed to freely write over the **temp registers t0-t6 (x5-x7, x28-x31)**.

Pass Arguments:  
a0-a7 and Stack

Move Stack Pointer

Save Saved Registers  
s0-s11

Don't Save Temp Registers  
s0-s11

# RISC-V Calling Convention

- The **caller** puts the **return address** (**PC+4**) in the **ra** register (**x1**)
  - The **JAL** command, returns from the function call.
- The **PC** is updated to the branch target and the function is run.

Upon finishing the subroutine:

- The **callee** places small return values in **a0-a1**.  
If larger, then in memory.
- The **callee saved** registers are restored (popped) from the **stack**.
- A **RET** command is given (**JALR x0, x1, 0**) to restore the **PC**.
- If the caller saved any registers, they are popped from the **stack** upon returning.

Store Return Address:

Ra (JAL)

Prepare Return Values

a0-a1 or memory

Restore Saved Registers

s0-s11

Return from subroutine

RET



# RISC-V Registers (ABI)

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function Arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

32 Registers is plenty but not too many.

For comparison, ARM-32 has only 16 and x86-32 has only 8!

Motivation

Basic RISC-V

Tidbits

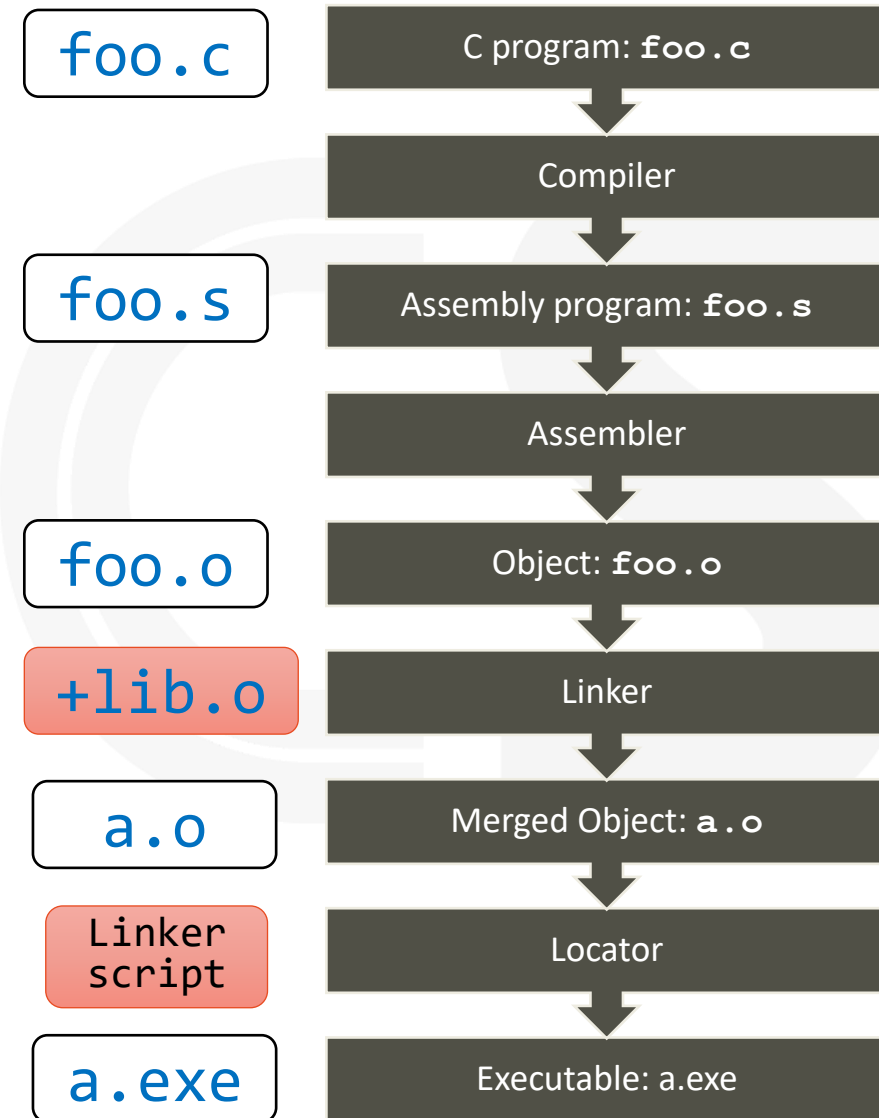
Procedure  
Calls

RISC-V CALL

# CALL in RISC-V

# Reminder: The Compilation Toolchain

- Start with a **C program**
- **Compile** it to an **Assembly program**
- **Assemble** it to an **Object file**
- **Link** all the assembly code into a **merged object file**
- Put in absolute addresses using the **linker script**
- **Load** the executable into the CPU and start running



# Compiled Hello.c: Hello.s

```
#include <stdio.h>

int main() {
    printf("Hello, %s\n",
           "world");

    return 0;
}
```

```
.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"
```

# Directive: enter text section  
# Directive: align code to 2^2 bytes  
# Directive: declare global symbol main  
# label for start of main  
# allocate stack frame  
# save return address  
# compute address of  
# string1  
# compute address of  
# string2  
# call function printf  
# restore return address  
# deallocate stack frame  
# load return value 0  
# return  
# Directive: enter read-only data section  
# Directive: align data section to 4 bytes  
# label for first string  
# Directive: null-terminated string  
# label for second string  
# Directive: null-terminated string

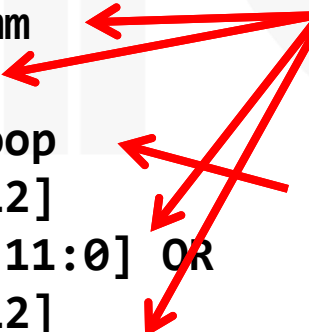
# Pseudo Instructions

- The compiler creates some pseudo-instructions
  - Instructions that assembler understands but not in machine
  - For example: `MV t1,t2` → `ADDI t1,t2,0`
- Assembler translates pseudo instructions to real instructions

```
MV    t0, t1
NEG    t0, t1
LI     t0, imm
NOT    t0, t1
BEQZ   t0, loop
LA     t0, str
```

```
ADDI    t0,t1,0
SUB     t0, zero, t1
ADDI    t0, zero, imm
XORI    t0, t1, -1
BEQ     t0, zero, loop
LUI     t0, str[31:12]
ADDI    t0, t0, str[11:0] OR
AUIPC   t0, str[31:12]
ADDI    t0, t0, str[11:0]
```

**DON'T FORGET:**  
sign extended  
immediates  
+  
Branch immediates  
count halfwords



# Producing Machine Language (Assembler)

- **Simple Case: Arithmetic, Logical, Shifts, and so on.**
  - All necessary info is within the instruction already
- **What about Branches and Jumps?**
  - PC-Relative (e.g., **BEQ**/**BNE** and **JAL**)
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch/jump over

- **Forward Referencing**

- Branches can refer to labels that are “forward” in the program:
- Solved by taking two passes over the program
  - **First pass** remembers position of labels
  - **Second pass** uses label positions to generate code

```
ADDI    t2,    zero,    9    # t2 = 9
L1:  SLT    t1,    zero,    t2    # 0 < t2? Set t1
      BEQ    t1,    zero,    L2    # NO! t2<=0;Go to L2
      ADDI    t2,    t2,    -1    # YES! t2 > 0; t2--
      J     L1    # Go to L1
```

**L2:**

3 words forward  
(6 halfwords)

3 words back  
(6 halfwords)

# Producing Machine Language (Assembler)

- **What about PC-relative jumps (**JAL**) and branches (**BEQ**, **BNE**)?**
  - **J offset** *pseudo instruction* expands to **JAL zero, offset**
  - Just count the number of instruction *halfwords* between target and jump to determine the offset: *position-independent code (PIC)*
- **What about references to static data?**
  - **LA** gets broken up into **LUI** and **ADDI** (use **AUIPC/ADDI** for PIC)
  - These require the full 32-bit address of the data
- **These can't be determined yet, so we create two tables:**
  - **Symbol Table**: labels and data that may be accessed across files
  - **Relocation Table**: absolute labels and data in the static section



# Assembler output: Linkable .o file

- **Object file header:**

- Size and position of the other pieces of the object file

- **Text segment:**

- The machine code

- **Data segment:**

- Binary representation of the static data in the source file

- **Relocation information:**

- Identifies lines of code that need to be fixed up later

- **Symbol table:**

- List of this file's labels and static data that can be referenced

- **Debugging information**

- **A standard format is ELF** [http://www.skyfree.org/linux/references/ELF\\_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)

```
00000000 <main>:
```

```
0: ff010113 ADDI    sp,sp,-16
```

```
4: 00112623 SW      ra,12(sp)
```

```
8: 00000537 LUI      a0,0x0      # addr placeholder
```

```
c: 00050513 ADDI      a0,a0,0      # addr placeholder
```

```
10: 000005b7 LUI      a1,0x0      # addr placeholder
```

```
14: 00058593 ADDI      a1,a1,0      # addr placeholder
```

```
18: 00000097 AUIPC     ra,0x0       # addr placeholder
```

```
1c: 000080e7 JALR      ra          # addr placeholder
```

```
20: 00c12083 LW       ra,12(sp)
```

```
24: 01010113 ADDI      sp,sp,16
```

```
28: 00000513 ADDI      a0,a0,0
```

```
2c: 00008067 JALR      ra
```

# Linker (and Locator)

- **Step 1:**

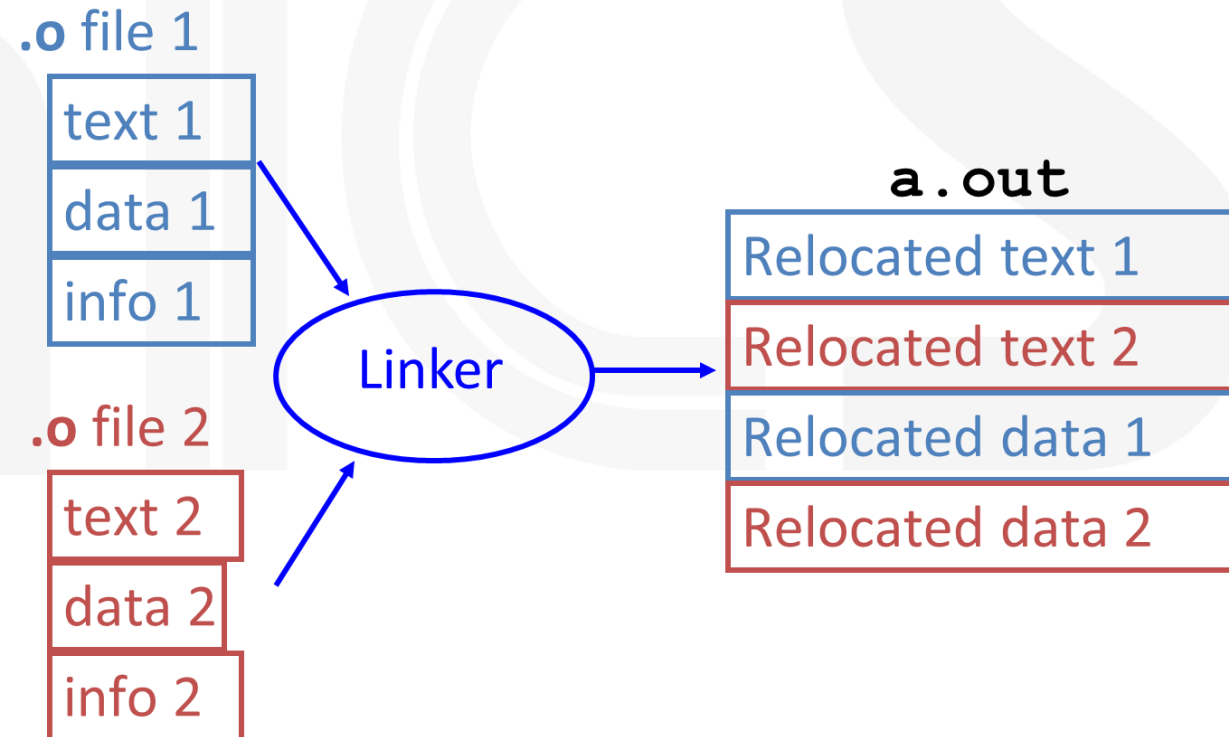
- Take text segment from each `.o` file and put them together

- **Step 2:**

- Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments

- **Step 3 (Locator):**

- Resolve references
- Go through Relocation Table and fill in all absolute addresses



# Linker

- Linker assumes first word of first text segment is at address `0x10000` for RV32.
- Linker knows:
  - Length of each text and data segment
  - Ordering of text and data segments
- Linker calculates:
  - Absolute address of each label to be jumped to (internal or external) and each piece of data being referenced
- To resolve references:
  - Search for reference (data or label) in all “user” symbol tables
  - If not found, search library files (e.g., for `printf`)
  - Once absolute address is determined, fill in the machine code appropriately
- Output of linker:
  - Executable file containing text and data (plus header)

# Linked Hello.o: a.out

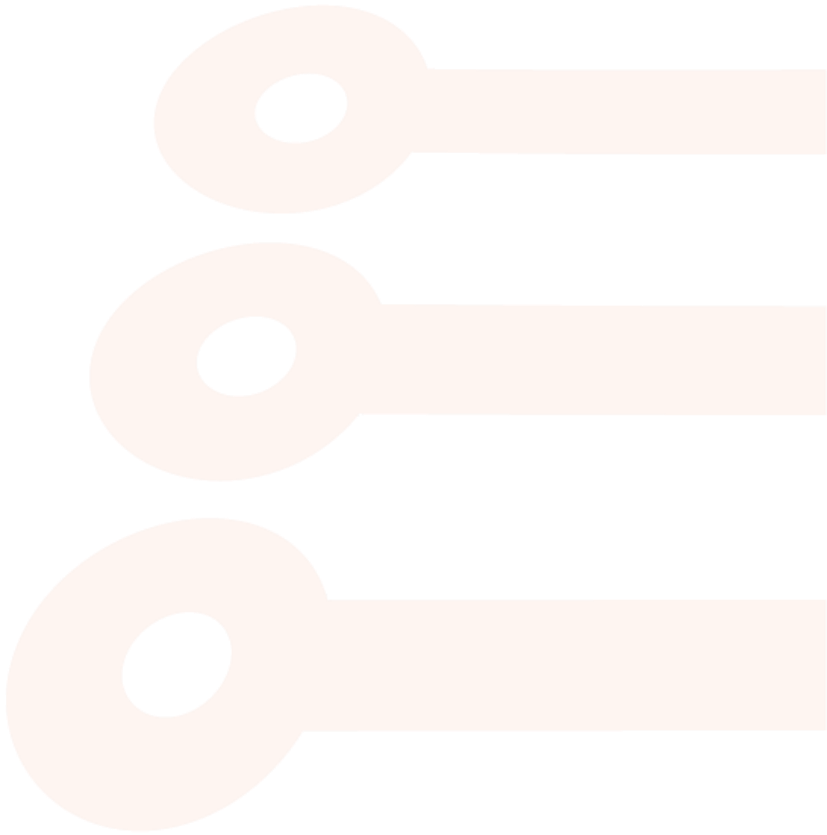
000101b0 <main>:

```
101b0: ff010113 ADDI sp,sp,-16
101b4: 00112623 SW ra,12(sp)
101b8: 00021537 LUI a0,0x21
101bc: a1050513 ADDI a0,a0,-1520 # 20a10 <string1>
101c0: 000215b7 LUI a1,0x21
101c4: a1c58593 ADDI a1,a1,-1508 # 20a1c <string2>
101c8: 288000ef JAL ra,10450 # <printf>
101cc: 00c12083 LW ra,12(sp)
101d0: 01010113 ADDI sp,sp,16
101d4: 00000513 ADDI a0,0,0
101d8: 00008067 JALR ra
```

# Loader

- Reads executable file's header to determine size of text and data segments
- Creates new address space for program
  - Large enough to hold `text` and `data` segments, along with a `stack` segment
- Copies instructions + data from executable file into the new address space
- Copies arguments (`argv`, `argc`) passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but `SP` assigned address of 1<sup>st</sup> free stack location
- Jumps to start-up routine (`main`)
  - Copies program's arguments from stack to registers & sets the `PC`
  - If `main` routine returns, start-up routine terminates program with `exit` system call

# Time to Kahoot!



ADVLSI

[www.kahoot.com](https://www.kahoot.com)

**ADVLSI Lecture 3b**

# References

- **Patterson, Hennessy “Computer Organization and Design – The RISC-V Edition”**
  - Chapters 2, 3
- **Patterson, Waterman “The RISC-V Reader”**
  - Chapters 1, 2
- **Berkeley CS-61C, “Great Ideas in Computer Architecture”**
  - Lecture 8, Spring 2015, Krste
  - Lecture 17, Fall 2018, Dan Garcia
- **RISC-V Spec**
- **Harry H. Porter “RISC-V: An Overview of the ISA”**
- **Krste Asanovic, Hot Chips Tutorial on RISC-V, Aug. 2019**