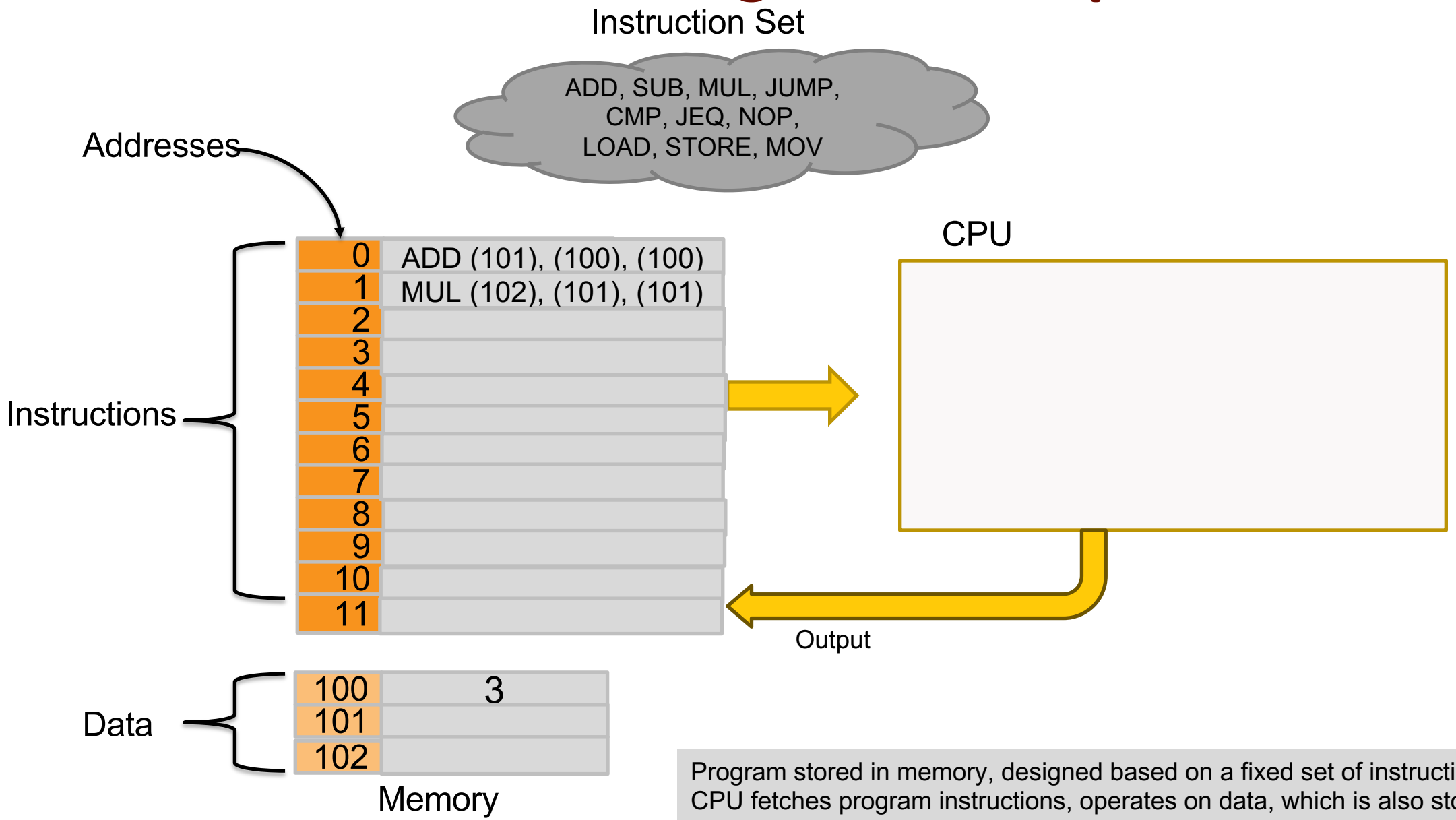
Instruction Set Architecture (Part 1)

Support for Programming Languages

Chester Rebeiro
Indian Institute of Technology Madras

chester@cse.iitm.ac.in

Stored Program Concept

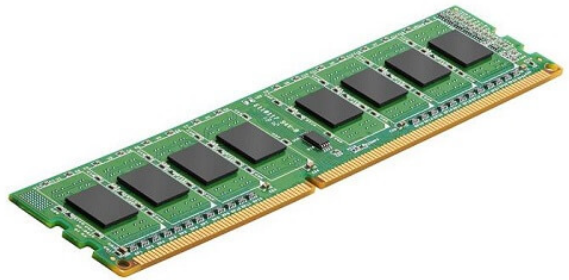


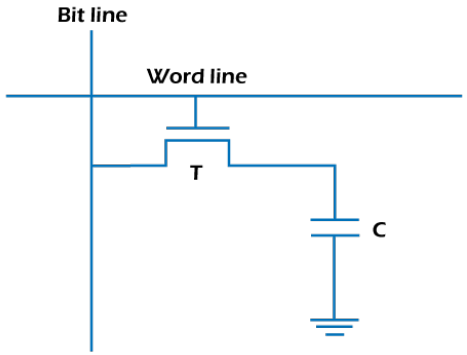
Program stored in memory, designed based on a fixed set of instructions
CPU fetches program instructions, operates on data, which is also stored in memory

Memory (RAM)

Large	Slow

- Varies from 1MB to 1TB
- Made with capacitors
- Used to store programs and data
- Accessed with addresses
  - Each address accesses a unique byte
  - 4 bytes is called a word



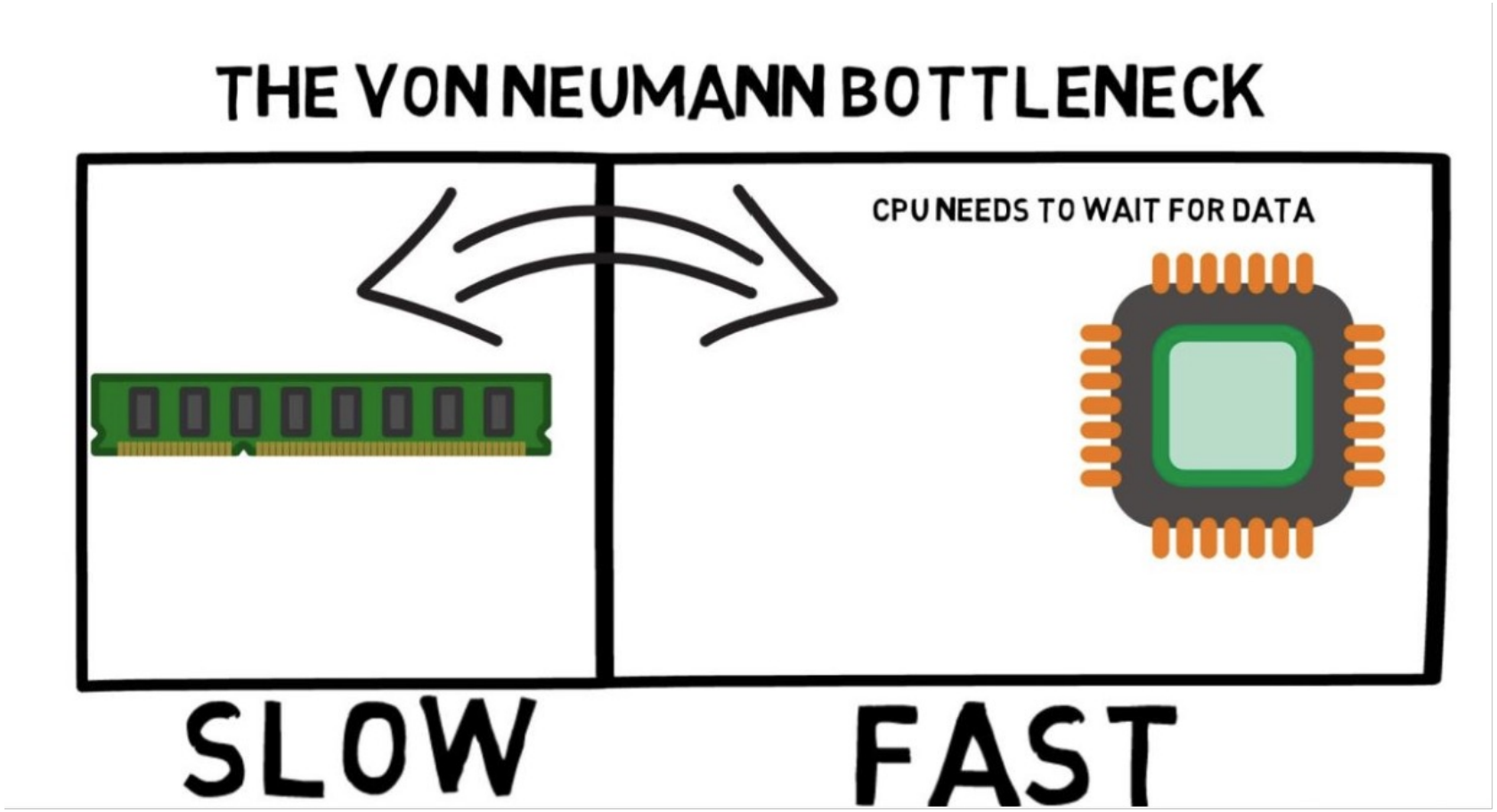


Byte at address A5

A0	A1	A2	A3
A4	A5	A6	A7
A8	A9	A10	A11
A12	A13	A14	A15
A16	A17	A18	A19
A20	A21	A22	A23
A24	A25	A26	A27
A28	A29	A30	A31

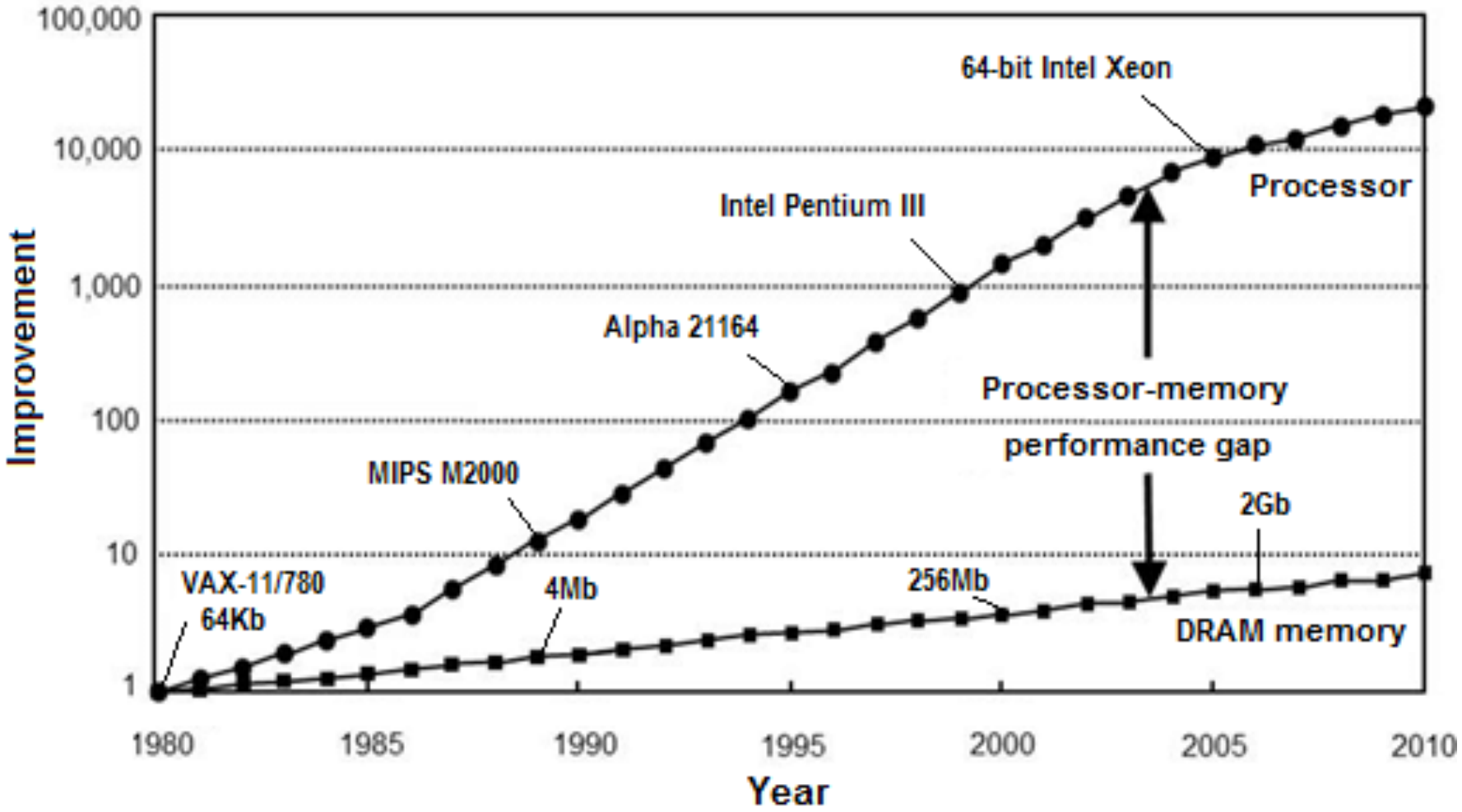
Word at address A5

VonNeumann Bottleneck



CPU too fast compared to memory. CPU

Processor-Memory Gap

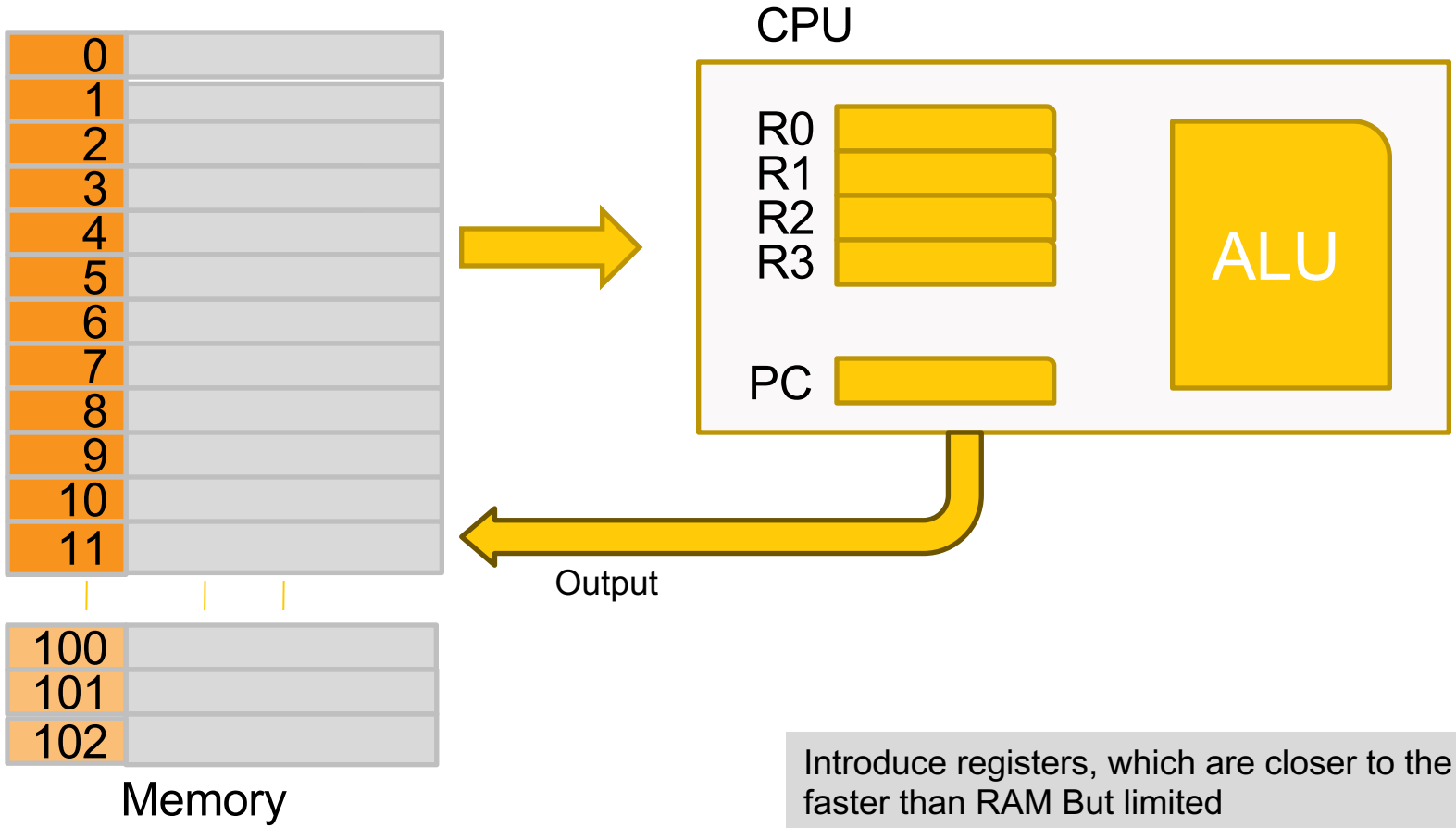


The gap has increased steadily

Registers

Instruction Set

ADD, SUB, MUL, JUMP,
CMP, JEQ, NOP,

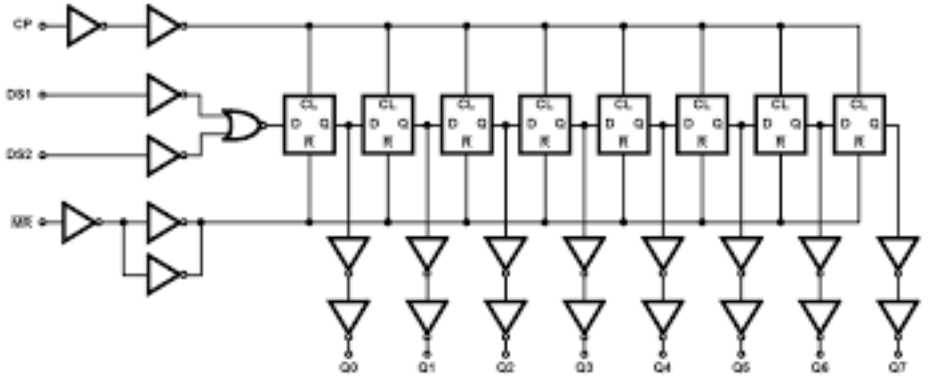


Introduce registers, which are closer to the CPU and much faster than RAM But limited

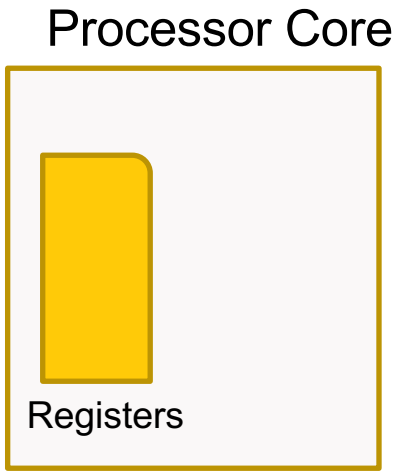
Registers

Few	Fast

- Typically around 16, 32, or 64 registers
  - How to select the number of register?
    - Fewer is faster (easier to decode, shorter wires between connections)
    - Too few is slower ☹ (need to go to memory to access data. Can't cache much)
    - Need to find the right number
- Placed very close to the processor.
  - Operate as fast as the processor.
  - Made with CMOS transistors



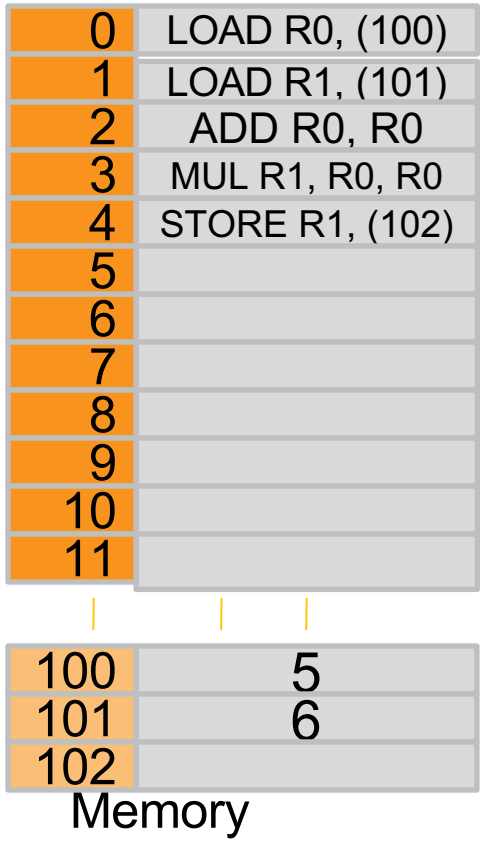
DRAMs vs Registers



Load / Store Instructions

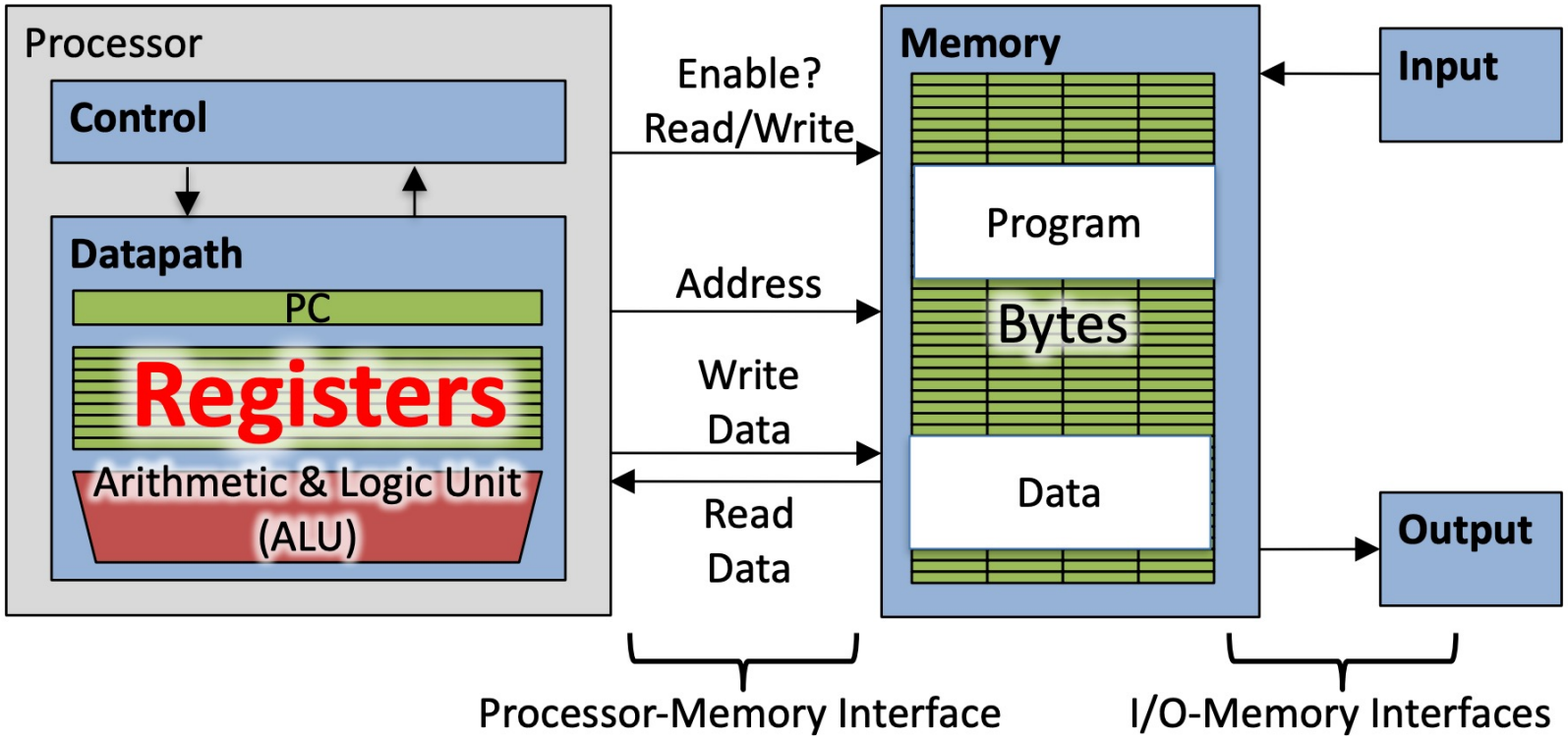
Instruction Set

ADD, SUB, MUL, JUMP,
CMP, JEQ, NOP,
LOAD, STORE, MOV



LOAD and STORE instructions to move data between memory and registers. All data now needs to be loaded first and then

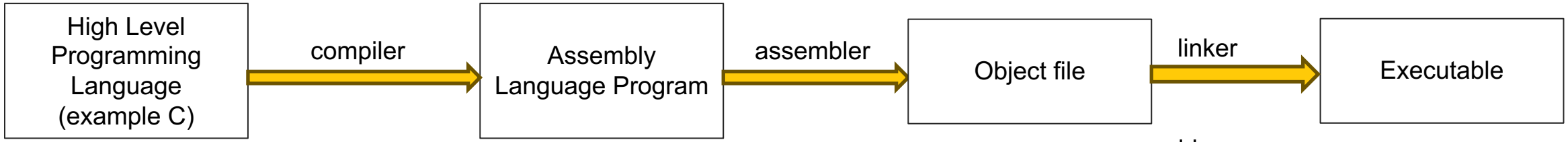
Interfaces



A slightly deeper look inside the processor
Notice the Processor-Memory Interface and I/O Interfaces

Taken from Krste Asanović and Randy H. Katz, CS61C slides.

Program transformations




```c
3  void increment_elements(int *array, int n){
4      int i;
5      for(i=0; i<n; ++i){
6          array[i] = array[i] + 1;
7      }
8  }
9
10 void exor_elements(int *array, int n){
11     int i;
12     for(i=0; i<n; ++i){
13         array[i] = array[i] ^ 0xFFFFFFFF ;
14     }
15 }
16
17 int main(){
18     int A[10];
19     int i;
20     int choice;
21
22     for(i=0; i<10; ++i) scanf("%d", &A[i]);
23     if (A[0] == 0 && A[1]==0){
24         increment_elements(&A[2], 8);
25     }
26     else{
27         exor_elements(&A[2], 8);
28     }
29 }
```

C Code

```
addiw    s0,s0,-16
unimp
ld       s0,32(a2)
lui      t1,0xffffa
bgeu     zero,a6,14
c.slli   zero,0xa
unimp
addi     s1,sp,32
lui      tp,0xfffe1
fld      fa2,376(sp)
ld       a3,80(a0)
0x7032
lw       a2,120(a4)
addiw    tp,tp,-5
fld      fa2,224(s0)
0x5f307032615f
fld      ft4,120(sp)
fld      fa2,224(s0)
0x5f307032645f
0x30703263
```

Assembly code

```
0000 797122f4 0018233c a4fcae87 232af4fc
0010 232604fe 35a08327 c4fe8a07 033784fd
0020 ba979843 8327c4fe 8a078336 84fdb697
0030 05270127 98c38327 c4fe8527 2326f4fe
0040 0327c4fe 832744fd 01278127 e345f7fc
0050 01000100 22744561 82807971 22f40018
0060 233ca4fc ae87232a f4fc2326 04fe05a8
0070 8327c4fe 8a070337 84fdba97 94438327
0080 c4fe8a07 033784fd ba973687 1347f7ff
0090 012798c3 8327c4fe 85272326 f4fe0327
00a0 c4fe8327 44fd0127 8127e343 f7fc0100
00b0 01002274 45618280 397106fc 22f88000
00c0 232604fe 2da01307 04fc8327 c4fe8a07
00d0 ba97be85 b7070000 13850700 97000000
00e0 e7800000 8327c4fe 85272326 f4fe8327
00f0 c4fe1b87 0700a547 e3d7e7fc 832704fc
0100 91ef8327 44fc99eb 930704fc a107a145
0110 3e859700 0000e780 000011a8 930704fc
0120 a107a145 3e859700 0000e780 00008147
```

Object file
(relocatable. Addresses start from 0)

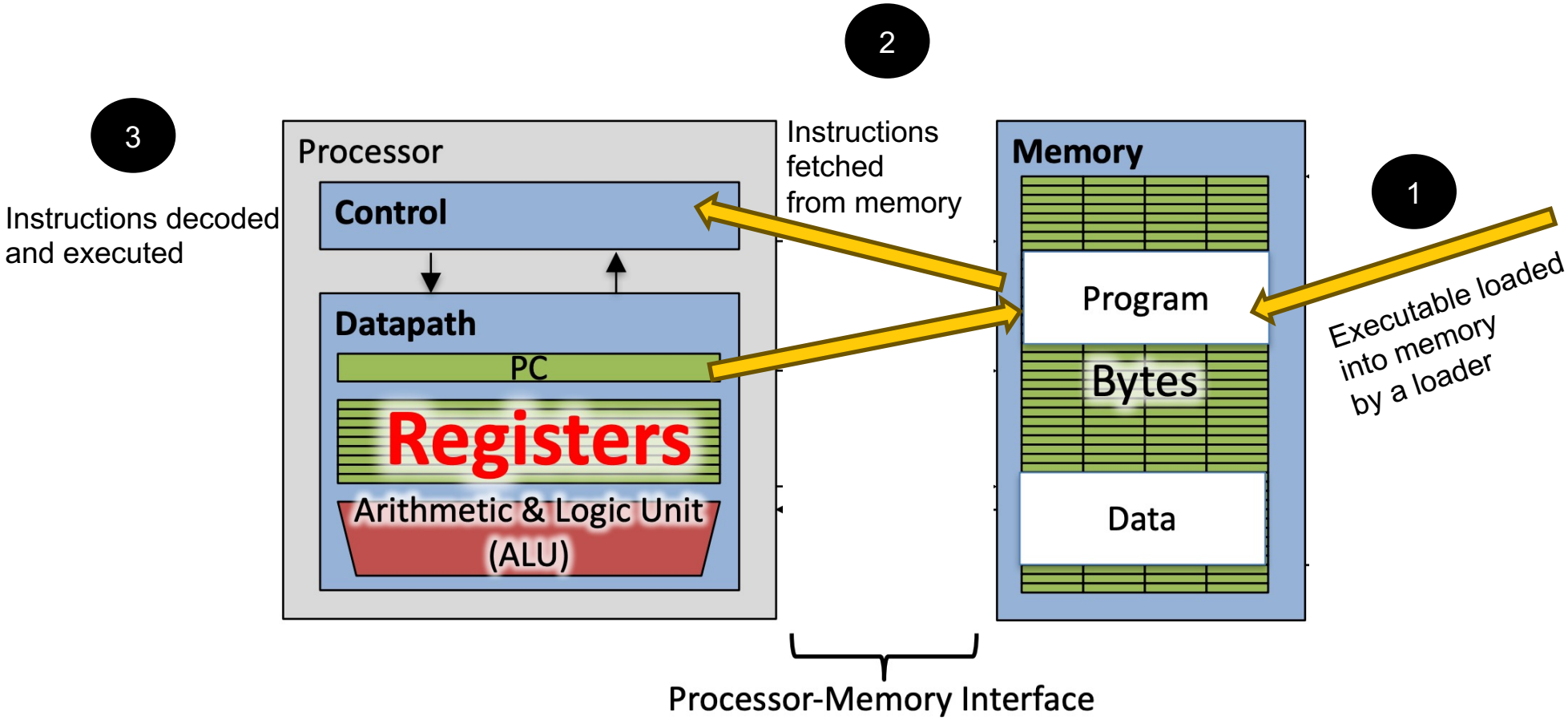
addresses

```
100b0 93070000 99c71735 00001305 e5aa6f20
100c0 30008280 97410100 9381c1ed 1385a176
100d0 17460100 1306066d 098e8145 ef000023
100e0 17250000 1305057e 19c51735 00001305
100f0 a5a7ef20 e07cef00 c01a0245 2c000146
10100 ef00e010 41a24111 22e01384 017a8347
10110 040006e4 99ef9307 000089cb 17350100
10120 1305c53e 97000000 e7000000 85472300
10130 f400a260 02644101 82809307 000099cb
10140 9385817a 17350100 1305453c 17030000
10150 67000000 82807971 22f40018 233ca4fc
10160 ae87232a f4fc2326 04fe35a0 8327c4fe
10170 8a070337 84fdba97 98438327 c4fe8a07
10180 833684fd b6970527 012798c3 8327c4fe
10190 85272326 f4fe0327 c4fe8327 44fd0127
101a0 8127e345 f7fc0100 01002274 45618280
101b0 797122f4 0018233c a4fcae87 232af4fc
101c0 232604fe 05a88327 c4fe8a07 033784fd
101d0 ba979443 8327c4fe 8a070337 84fdba97
```

Executable
(addresses resolved)

Encoded instructions

Program Execution



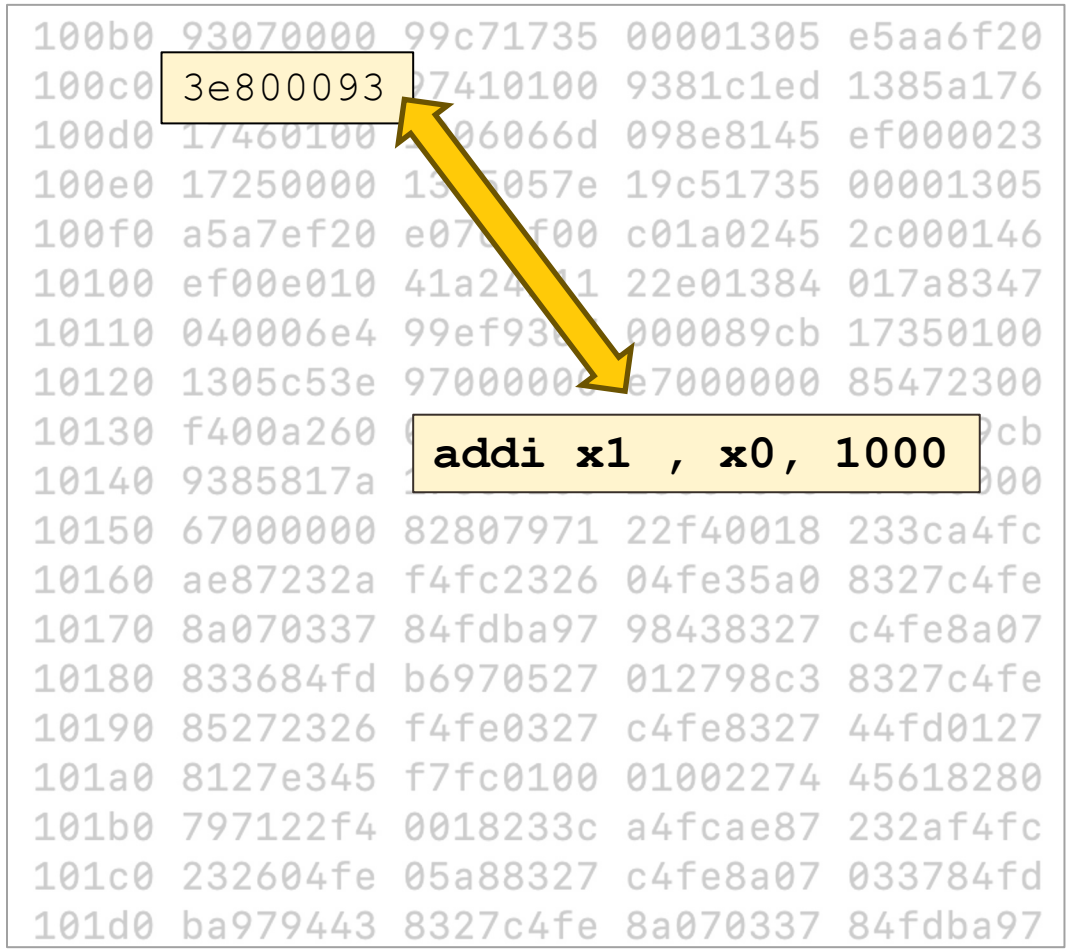

```
100b0 93070000 99c71735 00001305 e5aa6f20
100c0 30008280 97410100 9381c1ed 1385a176
100d0 17460100 1306066d 098e8145 ef000023
100e0 17250000 1305057e 19c51735 00001305
100f0 a5a7ef20 e07cef00 c01a0245 2c000146
10100 ef00e010 41a24111 22e01384 017a8347
10110 040006e4 99ef9307 000089cb 17350100
10120 1305c53e 97000000 e7000000 85472300
10130 f400a260 02644101 82809307 000099cb
10140 9385817a 17350100 1305453c 17030000
10150 67000000 82807971 22f40018 233ca4fc
10160 ae87232a f4fc2326 04fe35a0 8327c4fe
10170 8a070337 84fdba97 98438327 c4fe8a07
10180 833684fd b6970527 012798c3 8327c4fe
10190 85272326 f4fe0327 c4fe8327 44fd0127
101a0 8127e345 f7fc0100 01002274 45618280
101b0 797122f4 0018233c a4fcae87 232af4fc
101c0 232604fe 05a88327 c4fe8a07 033784fd
101d0 ba979443 8327c4fe 8a070337 84fdba97
```

Executable

Taken from Krste Asanović and Randy H. Katz, CS61C slides.

Instructions in the Program

Program Memory



- Each 32-bit in the program memory corresponds to an encoded instruction
- Unique map from instruction to encoding
- Instructions picked from the processor's Instruction Set Architecture

The Instruction Set

The language that CPUs understand

Different Instruction Set: x86, ARM, **RISC V**, MIPS.

- Program written for one processor cannot execute on another
- Early trend: add more instructions to new CPUs for elaborate operations
- RISC– Reduced Instruction Set Computing –
  - Instructions small and simple
  - Software does complicated operations by composing simpler ones

Any instruction set works over two aspects.

- Set of operations (opcodes)
- Set of registers

RISC V Instruction Set

- Fifth generation of RISC design from UC Berkeley
- License-free, royalty-free RISC ISA specification
- Experiencing rapid uptake in both industry and academia
- Both proprietary and open-source core implementations
- Appropriate for all levels of computing system, from microcontrollers to supercomputers
- Standard maintained by non-profit RISC-V Foundation

RISC V General Purpose Registers

XLEN is 32 or 64 bit, corresponding to 32 and 64 bit processors respectively

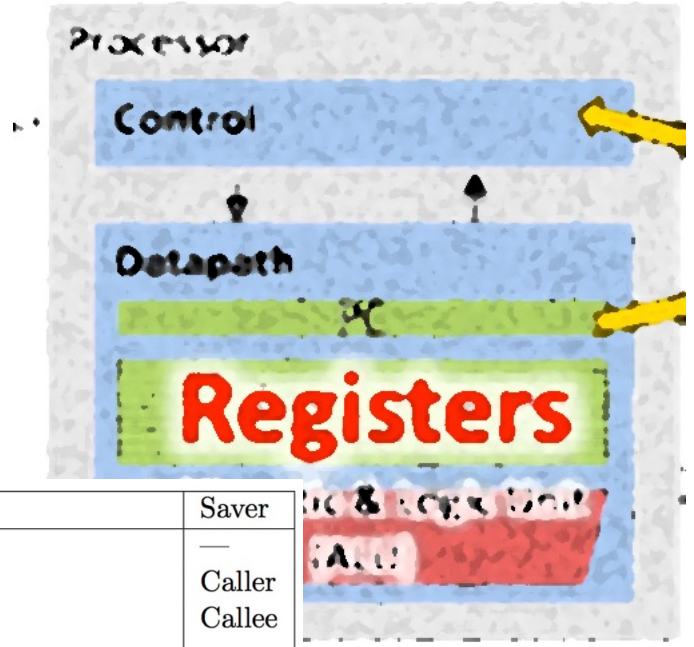
XLEN-1 … 0
x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

XLEN

Holds zero

XLEN-1 … 0
pc

XLEN



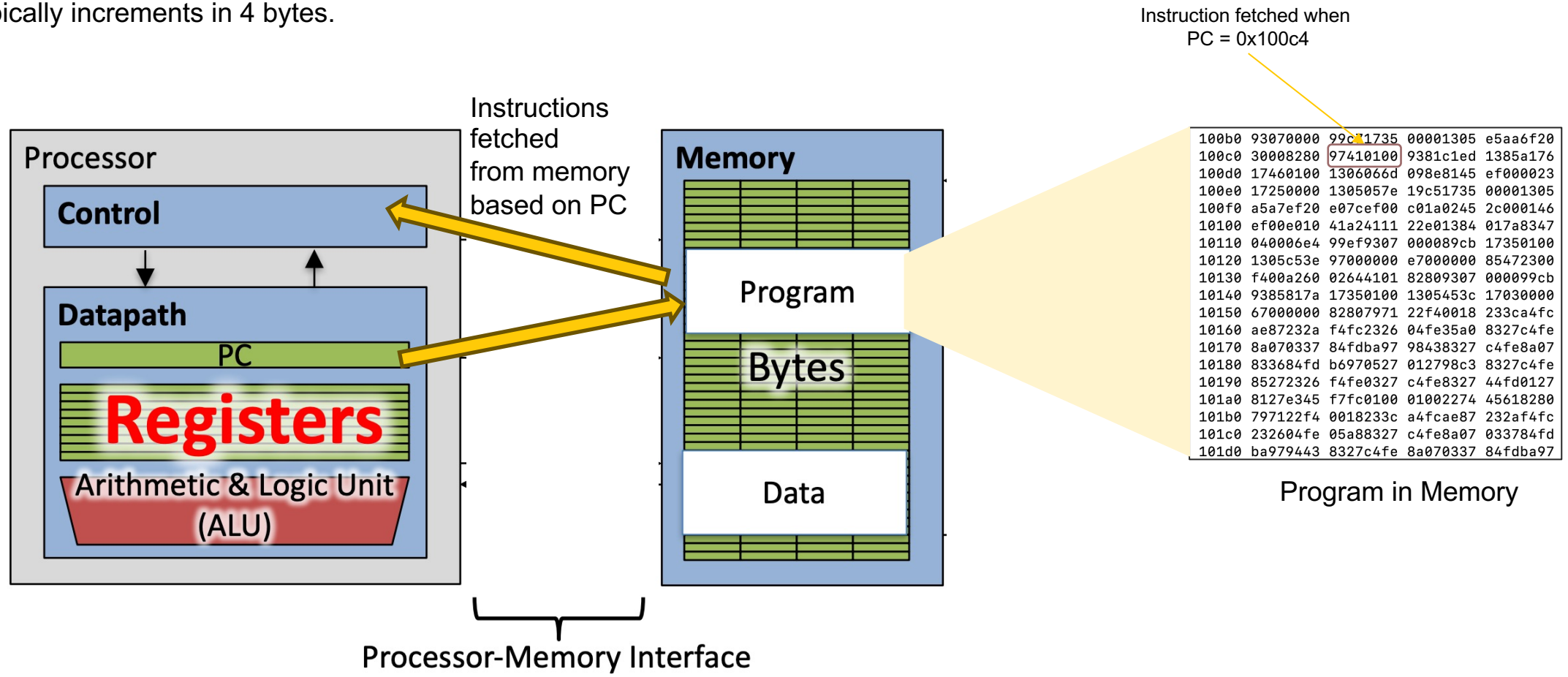
Numbers hardware understands

Human-friendly symbolic names in assembly code

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	t0	Temporary/alternate link register	Caller
x6–7	t1–2	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

The Program Counter (pc)

PC: Points to the current instruction being executed
Typically increments in 4 bytes.



Modified from Krste Asanović and Randy H. Katz, CS61C slides.

Programs

```
 3 void increment_elements(int *array, int n){
 4     int i;
 5     for(i=0; i<n; ++i){
 6         array[i] = array[i] + 1;
 7     }
 8 }
 9
10 void exor_elements(int *array, int n){
11     int i;
12     for(i=0; i<n; ++i){
13         array[i] = array[i] ^ 0xFFFFFFFF ;
14     }
15 }
16
17 int main(){
18     int A[10];
19     int i;
20     int choice;
21
22     for(i=0; i<10; ++i) scanf("%d", &A[i]);
23     if (A[0] == 0 && A[1]==0){
24         increment_elements(&A[2], 8);
25     }
26     else{
27         exor_elements(&A[2], 8);
28     }
29 }
```

Instructions for every program construct

```
 3 void increment_elements(int *array, int n){
 4     int i;
 5     for(i=0; i<n; ++i){
 6         array[i] = array[i] + 1;
 7     }
 8 }
 9
10 void exor_elements(int *array, int n){
11     int i;
12     for(i=0; i<n; ++i){
13         array[i] = array[i] ^ 0xFFFFFFFF ;
14     }
15 }
16
17 int main(){
18     int A[10];
19     int i;
20     int choice;
21
22     for(i=0; i<10; ++i) scanf("%d", &A[i]);
23     if (A[0] == 0 && A[1]==0){
24         increment_elements(&A[2], 8);
25     }
26     else{
27         exor_elements(&A[2], 8);
28     }
29 }
```

- Arithmetic operations
- Reading and Writing data (Data Transfer)
- Local variables
- Binary Operations
- Reading from I/0 (keyboard)
- Decision making operations
- Functional calls

Most common program constructs mapped to certain instructions

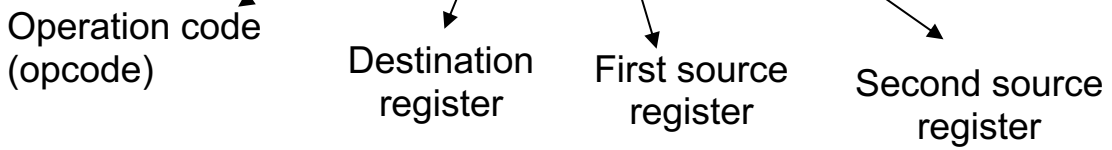
Typical instructions

C statement

```
a = b + c;
```

Assembly instruction with operands in registers

```
add x1, x2, x3    # x1 = x2 + x3
```



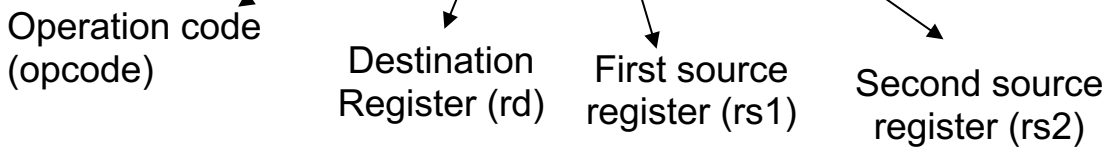
Typical instructions

C statement

```
a = b + c;
```

Assembly instruction with operands in registers

```
add x1, x2, x3    # x1 = x2 + x3
```

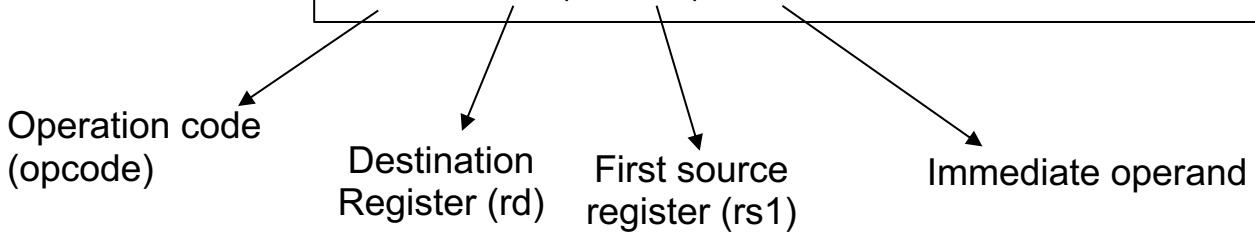


C statement

```
a = b - 10;
```

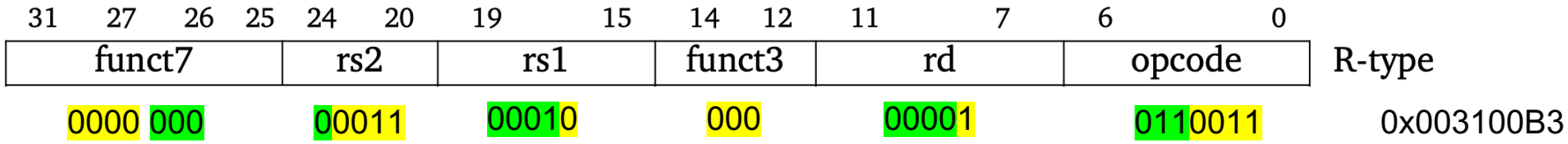
Assembly instruction with Immediate operands

```
addi x1, x2, -10    # x1 = x2 - 10
```



Instruction Encoding

```
add x1, x2, x3    # x1 = x2 + x3
```



funct7	rs2	rs1	funct3	rd	opcode	R-type
0000 000	00011	00010	000	00001	0110011	0x003100B3

```
add x1, x3, x2    # x1 = x3 + x2
```

funct7	rs2	rs1	funct3	rd	opcode	R-type
0000 000	00010	00011	000	00001	0110011	0x002180B3

```
sub x1, x3, x2    # x1 = x3 - x2
```

funct7	rs2	rs1	funct3	rd	opcode	R-type
0100 000	00010	00011	000	00001	0110011	0x402180B3

Ref.. RISCV_CARD.PDF

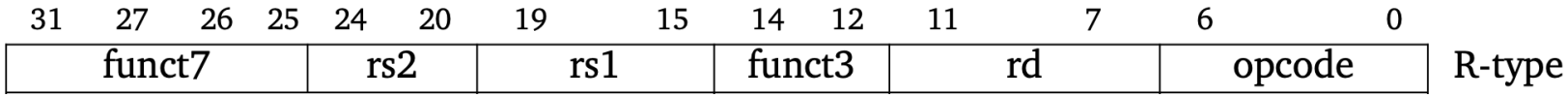
Try it out here..RISCV ONLINE ASSEMBLER

R-Type Instructions in RISC V

R-Type

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

Unsigned comparison



R-type

Shift Left Logical (shift left x12 by 2-bits)

```
sll x12, x12, x11     # x12 = x12 << x1

Initial values: x11 = 0x2, x12 = 0x1
Final values:   x12=0x4
```

Shift Right Logical (shift right x12 by 2-bits)

```
srl x12, x12, x11     # x12 = x12 >> x11

Initial values: x11 = 0x2, x12 = 0xabcd5043
Final values:   x12=0x2af35410
```

Shift Right Arithmetic (shift right x12 by 2-bits) and extend msb)

```
sra x12, x12, x11     # x12 = x12 >> x1

Initial values: x11 = 0x2, x12 = 0xabcd5043
Final values:   x12=0xeaf35410
```

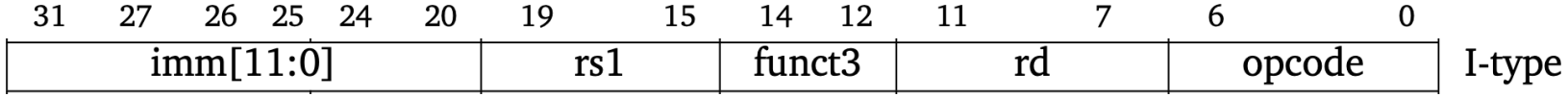
Try them out here: RISC V Online Interpreter

I-Type Instructions in RISC V

I-Type

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends

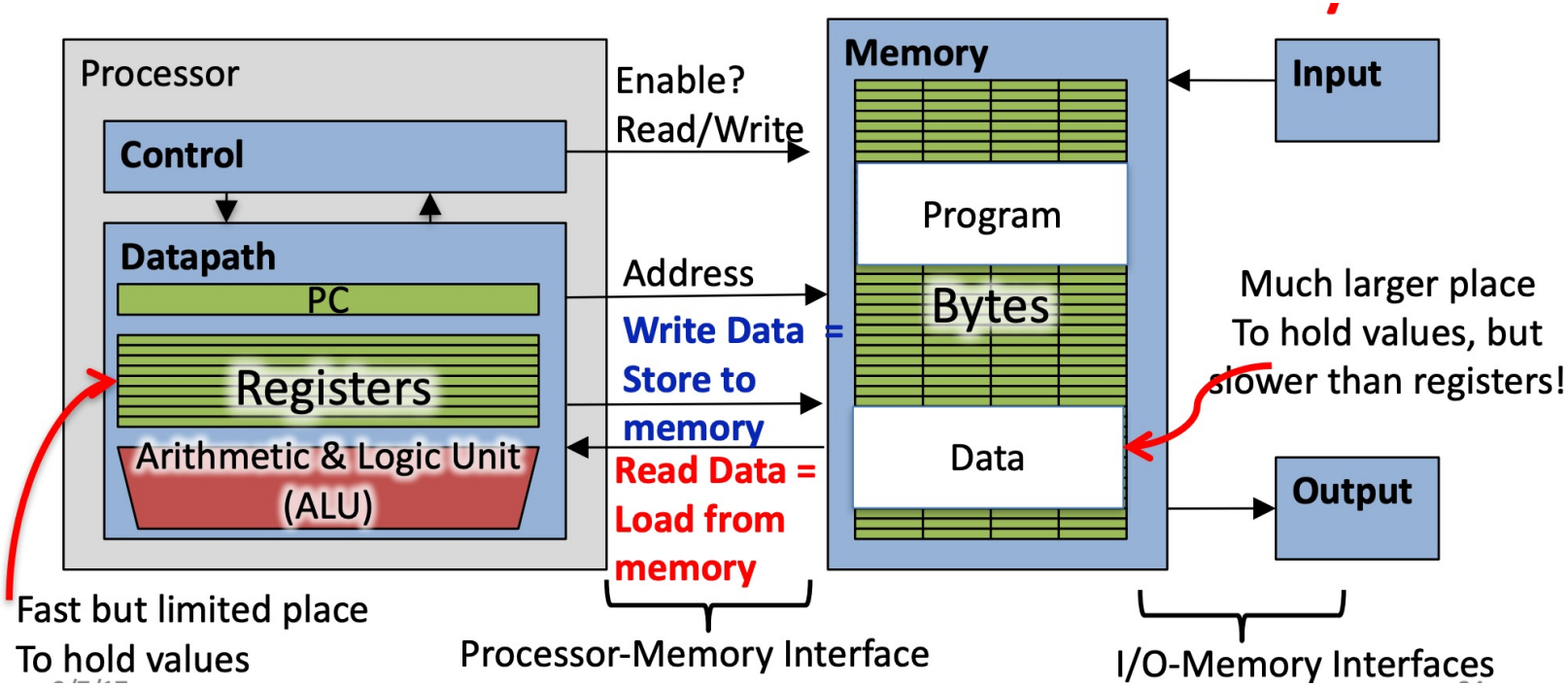
max shift is 16



Immediate is sign extended and is of 12 bits

```
addi x12, x12, 2047    # x12 = x12 + 0x000007FF
addi x13, x13, 2048    # x13 = x13 + 0xFFFFF800

Initial values: x12 = x13 = 0x1
Final values:   x12 = 2048
                x13 = -2047
```

Memory Load and Store Operations



Load : Read data from memory at the given address to a register
Store: Write data from a register to memory at the given address

Taken from Krste Asanović and Randy H. Katz, CS61C slides.

Instructions for Data Transfer

data transfer instruction

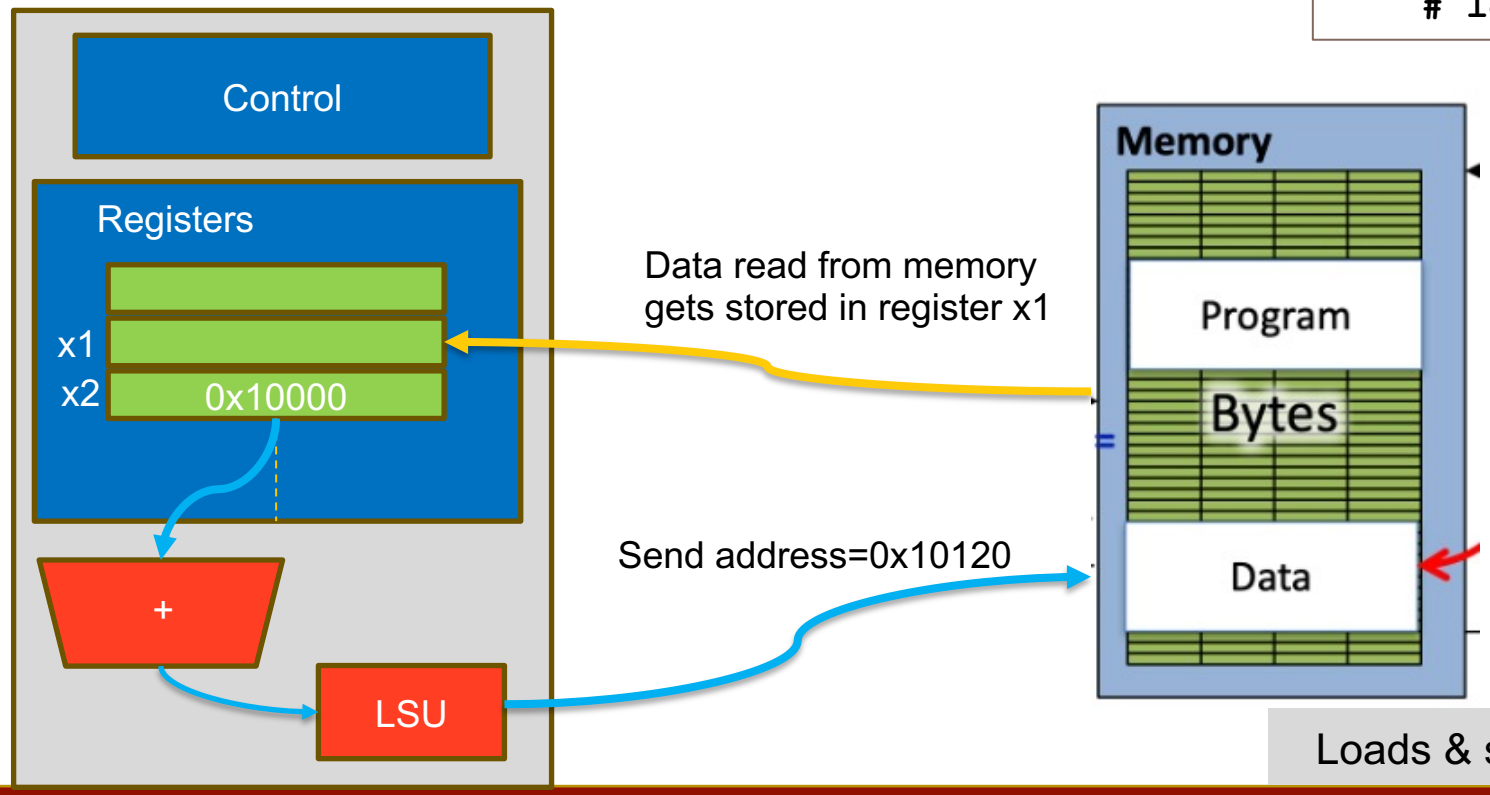
A command that moves data between memory and registers.

Load 32 bits into destination register rd from address rs1+imm; rs is the source register, imm a 12-bit signed immediate

```
lw rd, imm(rs1)
```

```
rd = M[rs1+imm][0:31]
```

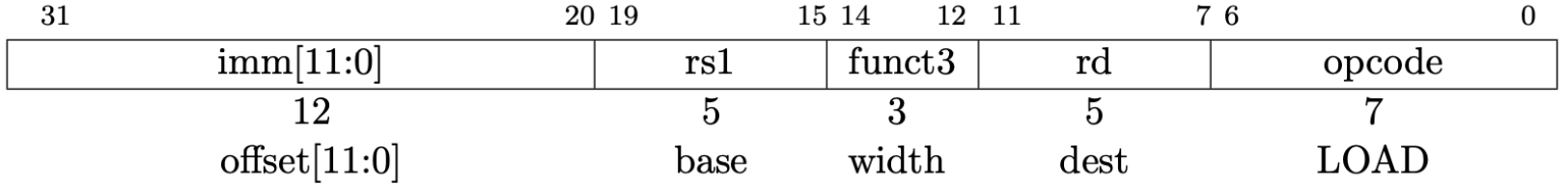
```
Example: lw x1,0x120(x2)
     # load into register x1 from x2+0x120
```



Loads & stores handled by the Load Store Unit (LSU)

Load Instruction Encoding

```
lw rd, imm(rs1)
```



- Imm is 12-bits and signed. Thus range of immediate is -2048 to +2047

C construct: arrays

```
int a[10];
int b = a[5];
```

dest: rd

base register: rs1

offset: imm

Equivalent assembly instruction

```
lw x10, 24(x12)
```

dest: rd

base of array

offset: imm

Instructions for Data Transfer

data transfer instruction

A command that moves data between memory and registers.

Load 32 bits into destination register rd from address rs+imm; rs is the source register, imm a 12-bit signed immediate

```
lw rd, imm(rs)
```

```
rd = M[rs+imm][0:31]
```

```
Example: lw x1,128(x2)
```

Instructions for Data Transfer

data transfer instruction

A command that moves data between memory and registers.

Load 32 bits into destination register rd from address rs+imm; rs is the source register, imm a 12-bit signed immediate

```
lw rd, imm(rs)
```

```
rd = M[rs+imm][0:31]
```

```
Example: lw x1,128(x2)
```

Load 8 bits into destination register rd from address rs+imm; rs is the source register, imm a 12-bit signed immediate
rs is sign extended

```
lb rd, imm(rs)
```

```
rd = M[rs+imm][0:7]
```

```
Example: lw x1,128(x2)
```

Load 8 bits into destination register rd from address rs+imm; rs is the source register, imm a 12-bit signed immediate
rs is zero extended

```
lbu rd, imm(rs)
```

```
rd = M[rs+imm][0:7]
```

```
Example: lbu x1,128(x2)
```

Similarly, `lh` and `lhu` are 16-bit loads that are equivalent to `lb` and `lbu`

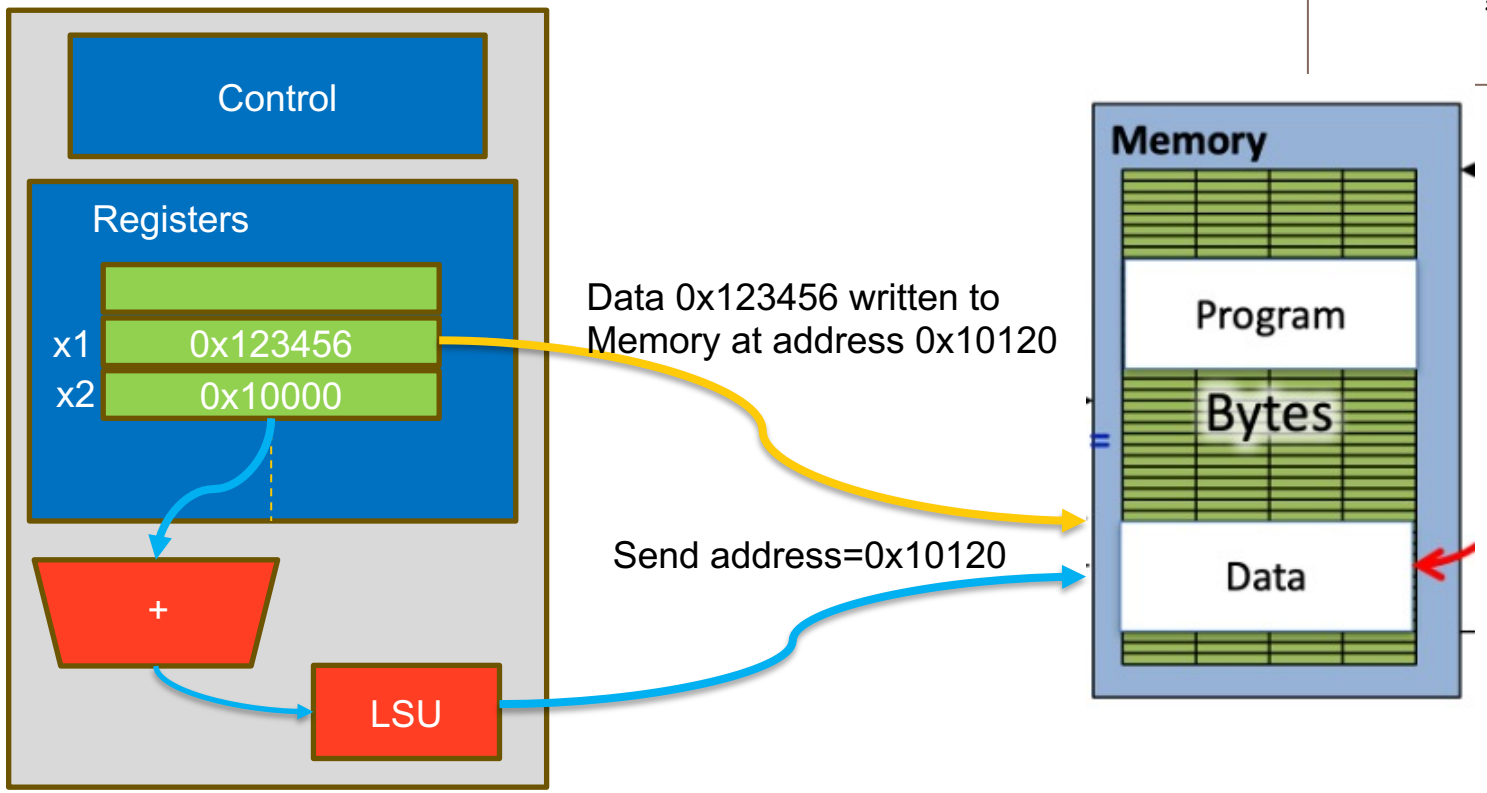
Store Instruction

Store (write) 32 bits into destination register rd from address rs1+imm; rs is the source register, imm a 12-bit signed immediate

```
sw rs2, imm(rs1)
```

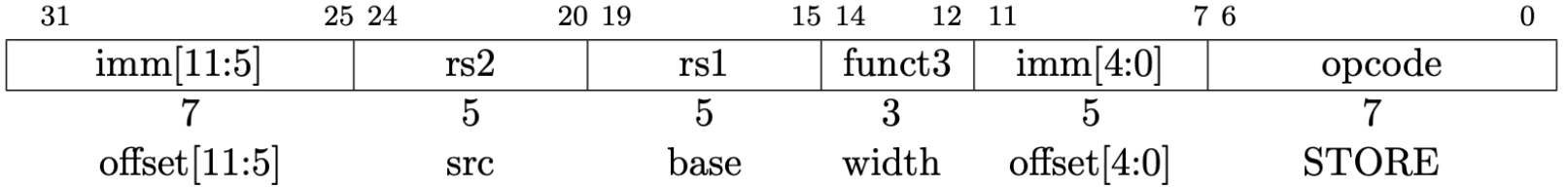
```
M[rs1+imm][0:31] = rs2
```

```
Example: sw x1,0x120(x2)
        # store into location x2 + 0x120
          the contents of x1
```



Store Instruction Encoding

```
sw rs2, imm(rs1)
```



- Imm is 12-bits and signed. Thus, range of immediate is -2048 to +2047

C construct: arrays

```
int a[10];
a[6] = b;
```

src: rs2

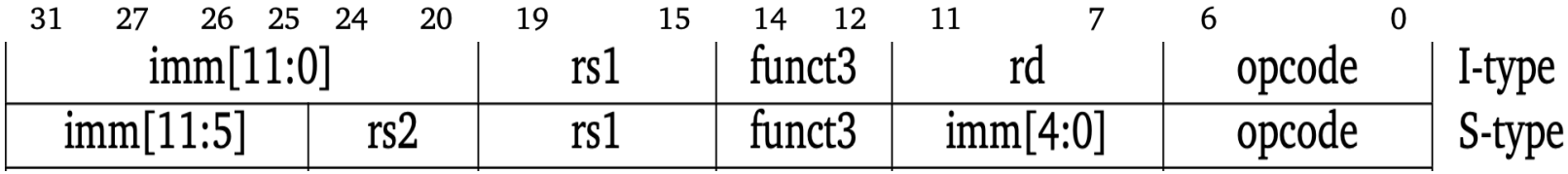
base register: rs1

offset: imm

Equivalent assembly instruction

```
sw x10, 24(x12)
```

Other Load and Store Instructions



Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
`lb`	Load Byte	I	`0000011`	`0x0`		`rd = M[rs1+imm][0:7]`	
`lh`	Load Half	I	`0000011`	`0x1`		`rd = M[rs1+imm][0:15]`	
`lw`	Load Word	I	`0000011`	`0x2`		`rd = M[rs1+imm][0:31]`	
`lbu`	Load Byte (U)	I	`0000011`	`0x4`		`rd = M[rs1+imm][0:7]`	zero-extends
`lhu`	Load Half (U)	I	`0000011`	`0x5`		`rd = M[rs1+imm][0:15]`	zero-extends
`sb`	Store Byte	S	`0100011`	`0x0`		`M[rs1+imm][0:7] = rs2[0:7]`	
`sh`	Store Half	S	`0100011`	`0x1`		`M[rs1+imm][0:15] = rs2[0:15]`	
`sw`	Store Word	S	`0100011`	`0x2`		`M[rs1+imm][0:31] = rs2[0:31]`	

Example load and store

```
addi    x1, x0, 0xABCDEF00
sw      x1, 4(x2)
lb      x3, 5(x2)
lbu     x4, 5(x2)
```

If x1 has 0xABCDEF00

After execution

x3 will hold, 0xFFFFFFEF

x4 will hold, 0x000000EF

Aligned vs Non-aligned memory accesses

For words

Loads and stores typically optimized for word aligned addresses. Multiple of 4 bytes.

Aligned load: From an address that is a multiple of word size (word size here is 4 bytes). Most often used.

```
x2 holds A16

lw x1, 0x0(x2)
```

Non-aligned load: From an address that is not a multiple of word size (word size here is 4 bytes). Not always supported in all architectures. Slow. May be more difficult to support due to atomicity.

```
x2 holds A6

lw x1, 0x0(x2)
```

Memory

A0	A1	A2	A3
A4	A5	A6	A7
A8	A9	A10	A11
A12	A13	A14	A15
A16	A17	A18	A19
A20	A21	A22	A23
A24	A25	A26	A27
A28	A29	A30	A31

Aligned vs Non-aligned memory accesses

For half words

Loads and stores for half words typically optimized for ½ word aligned addresses. Multiple of 2 bytes.

Aligned: From an address that is a multiple of ½ word size (word size here is 4 bytes). Most often used.

```
x2 holds A6

lhu x1, 0x0(x2)
```

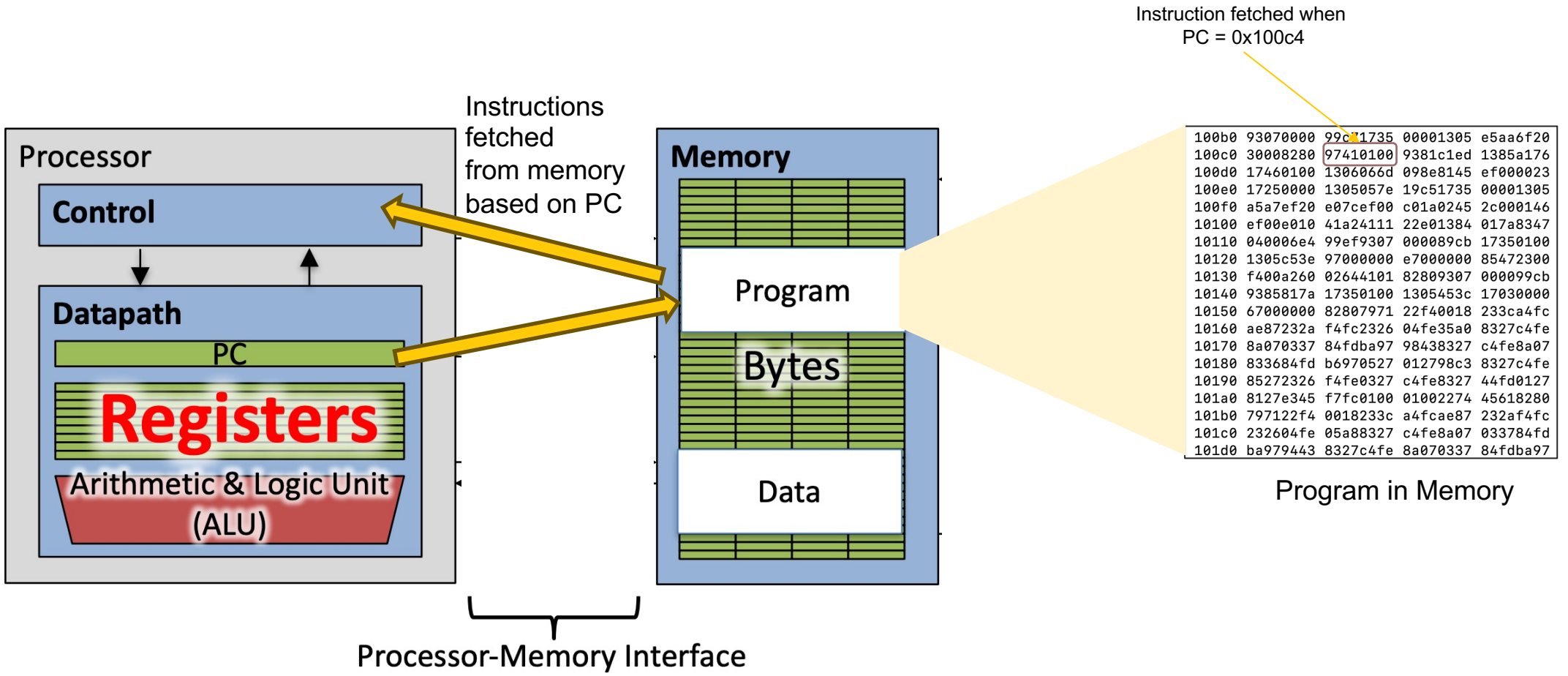
Non-aligned : From an address that is not a multiple of word size (word size here is 4 bytes). Not always supported in all architectures. Slow. May be more difficult to support due to atomicity.

```
x2 holds A17

lhu x1, 0x0(x2)
```

Memory

A0	A1	A2	A3
A4	A5	A6	A7
A8	A9	A10	A11
A12	A13	A14	A15
A16	A17	A18	A19
A20	A21	A22	A23
A24	A25	A26	A27
A28	A29	A30	A31

The Program Counter (pc)



PC: Points to the current instruction being executed Typically increments in 4 bytes. However, the sequence breaks due to branches.

Modified from Krste Asanović and Randy H. Katz, CS61C slides.

Conditional Branch Instructions

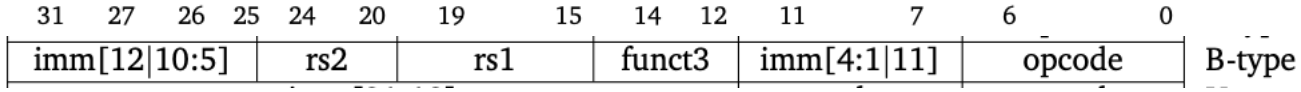
offset
with respect
to the PC
(PC relative addressing)

C conditional statement

```
if (a != b){
    // execute here if TRUE
}
```

```
Addr:
  bne  x4, x5, label
  // execute instructions
  // here if x4 == x5


label:
  . . .
```

Address to branch:
target=PC + label



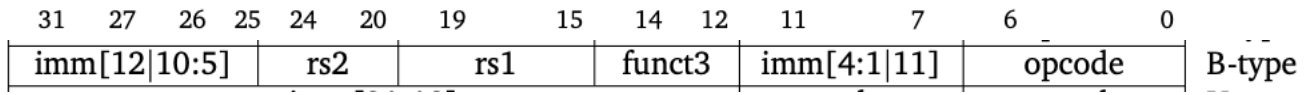
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends

Branch range: +/- 4KiB for beq, bne, blt, bge;
+8KiB for bltu and bgeu

Can only jump to even addresses.

Conditional Branch Instructions

```
Addr:
  bne  x4, x5, label
  // execute instructions
  // here if x4 == x5


label:
  . . .
  . . .
  . . .
```




```
If branch is not taken{
    PC = PC + 4
} else if branch is taken{
    PC = PC + 4 * Immediate
}
```

immediate is number of instructions to jump (remember, specifies words)
either forward (+) or backwards (–

Pseudo Instructions

- Easy to understand and use. They translate to base instructions
- Example 1: Copy register from rs1 to rd.
  Pseudo instruction **`mv rd, rs1`** translates to **`addi rd, rs1, 0`**
- Example 2: Load immediate into registers
  Pseudo instruction **`li rd, CONSTANT`**
- Example 3: Load address of symbol
  Pseudo instruction **`la rd, SYMBOL`**
  translates to

```
lui rd, SYMBOL[31:12]
addi rd, t0, SYMBOL[11:0]
```

RISC V Assembly Program Manual: https://shakti.org.in/docs/risc-v-asm-manual.pdf
RISC V Assembly Programmer's Manual: https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md
RISC V Programming Book: https://riscv-programming.org/book/riscv-book.html

First RISC V Assembly Program

Strlen C code

```
char str[]= "Hello World!"
int i;
for (i=0; str[i] != '\0'; i);
return i
```

strlen.S

```
.section .text              # represents the section for the code
.global strlen
.global _start

_start:                     # execution starts from this label
strlen:
    la      a1, str         # load address of str into a1
    li      t0, 0           # i = 0
1: # Start of for loop
    add     t1, t0, a1      # Add the byte offset for str[i]
    lb      t1, 0(t1)       # Dereference str[i]
    beqz    t1, 1f          # if str[i] == 0, break for loop
    addi    t0, t0, 1       # Add 1 to our iterator
    j       1b              # Jump back to condition (1 backwards)
1: # End of for loop
    mv      a0, t0          # Move t0 into a0 to return
    ret                     # Return back via the return address register

.data                       # represents the section for the data
str:   .ascii "Hello World!"
```

First RISC V Assembly Program

strlen.S

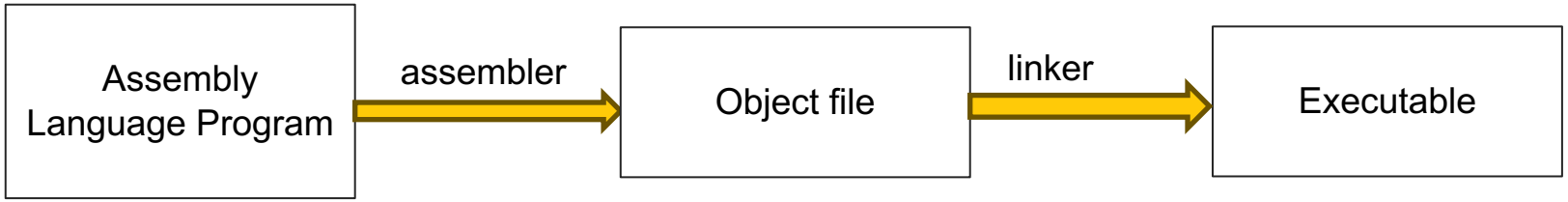
```
.section .text                  # represents the section for the code
.global strlen
.global _start


_start:                         # execution starts from this label
strlen:
    la      a1, str             # load address of str into a1
    li      t0, 0               # i = 0
1: # Start of for loop
    add     t1, t0, a1          # Add the byte offset for str[i]
    lb      t1, 0(t1)           # Dereference str[i]
    beqz    t1, 1f              # if str[i] == 0, break for loop
    addi    t0, t0, 1           # Add 1 to our iterator
    j       1b                  # Jump back to condition (1 backwards)
1: # End of for loop
    mv      a0, t0              # Move t0 into a0 to return
    ret                         # Return back via the return address register

.data                           # represents the section for the data
str:    .ascii "Hello World!"
```

Assembly Directives.
Starts with a '.' used to control the assembler

Labels are markers used to represent program locations

These are pseudo-instructions

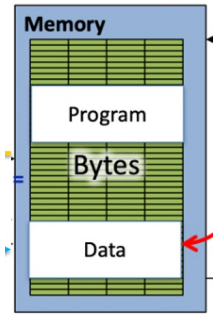
Create an Executable




```
riscv64-unknown-elf-gcc -nostdlib -nostartfiles -T ./spike.lds  strlen.S -o strlen.elf
```

Invokes the assembler

Linker Descriptor File
How should the executable be placed in memory?
The memory map:



Input assembly program

Output executable

The Linker Descriptor File

```
OUTPUT_ARCH("riscv")

ENTRY(_start)  #execution should start
               from this label
SECTIONS
{
    . = 0x80000000; #at address


    .text : ALIGN(4K) #place code section
    {
        *(.text)
        *(.text.init)
    }

    .data : ALIGN(4K) #place data section
    {
        *(.data)
    }

}
```

The Linker Descriptor File

```
OUTPUT_ARCH("riscv")

ENTRY(_start)  #execution should start
               from this label

SECTIONS
{
     . = 0x80000000; #at address


     .text : ALIGN(4K) #place code section
     {
          *(.text)
          *(.text.init)
     }

     .data : ALIGN(4K) #place data section
     {
          *(.data)
     }

}
```

Checking out the .text part of the executable

```
root@b32eeece94b2:/home/os-iitm/xv6-riscv# riscv64-unknown-elf-objdump -dj .text strlen.elf

strlen.elf:     file format elf64-littleriscv


Disassembly of section .text:

0000000080001000 <_start>:
    80001000:   00001597                auipc   a1,0x1
    80001004:   00058593                mv      a1,a1
    80001008:   4281                    li      t0,0
    8000100a:   00b28333                add     t1,t0,a1
    8000100e:   00030303                lb      t1,0(t1)
    80001012:   00030463                beqz    t1,8000101a <_start+0x1a>
    80001016:   0285                    addi    t0,t0,1
    80001018:   bfcd                    j       8000100a <_start+0xa>
    8000101a:   8516                    mv      a0,t0
    8000101c:   8082                    ret
```

The Linker Descriptor File

```
OUTPUT_ARCH("riscv")

ENTRY(_start)  #execution should start
               from this label

SECTIONS
{
     . = 0x80000000; #at address


     .text : ALIGN(4K) #place code section
     {
          *(.text)
          *(.text.init)
     }

     .data : ALIGN(4K) #place data section
     {
          *(.data)
     }

}
```

Checking out the .data part of the executable

```
root@b32eeece94b2:/home/os-iitm/xv6-riscv# riscv64-unknown-elf-objdump -sj .data strlen.elf

strlen.elf:     file format elf64-littleriscv

Contents of section .data:
 80002000 48656c6c 6f20576f 726c6421          Hello World!
```

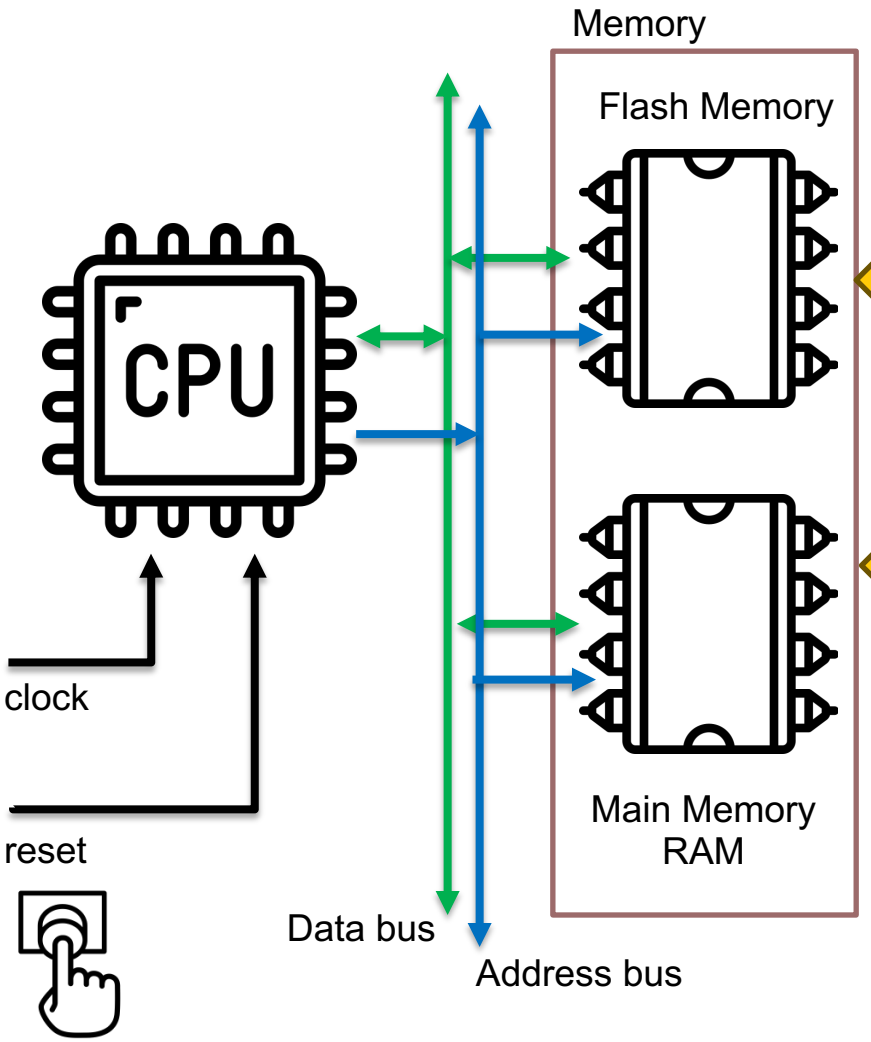
Executing the program

- With an Operating System
  - The OS loads the program into memory based on the memory map
  - It sets the program counter to address of entry (ie. _start)
  - Program executes

  (more on this later)

- Without an Operating System
  - Bare metal

Executing the program – Baremetal



Will hold the .text parts of program

Will hold the .data parts of program

On reset

- All registers set to 0
- PC set to reset vector (fixed address, for example 0x80000000)
- This address should be present in the flash memory, where the program to be executed must be present

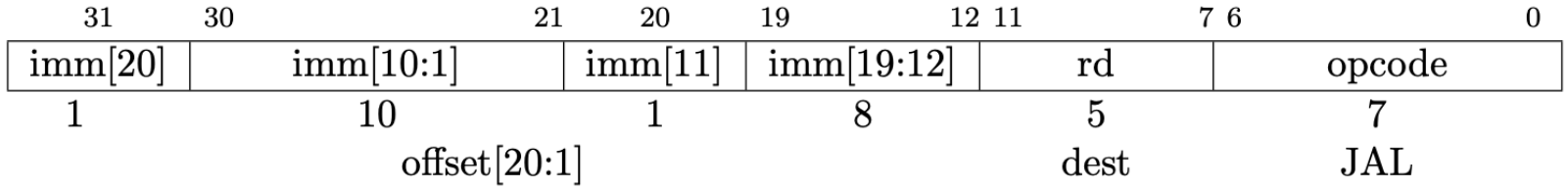
Support for Function Calls

- Invoking functions
- Returning from a function
- Passing parameters
- Managing local variables

Instructions to Invoke a Function Call

```
jal rd, imm
```

Jump to imm+PC, while at the same time store PC+4 in register rd



Imm is an offset, sign extended and added to PC.
Thus, all jumps are relative to PC.
Jumps can be +/- 1MB from the current PC position

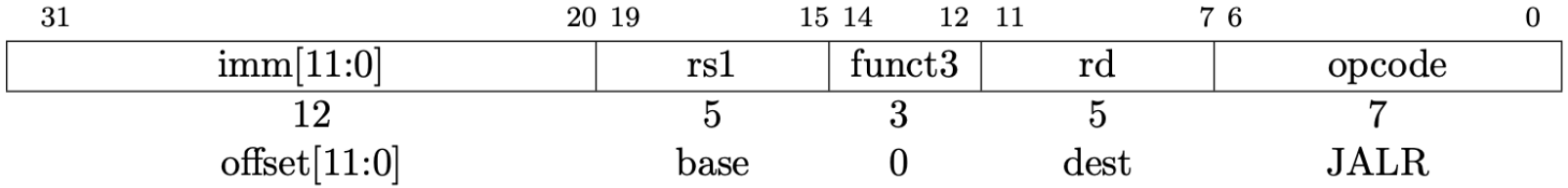
Pseudo instruction **`j label`** → realized as **`jal x0, label`**

RISC V Calling Convention

Indirection Jumps

```
jalr rd, rs1, imm
```

Indirect Jump to rs1+imm, while at the same time store PC+4 in register rd



Note. Jalr is not relative to the PC

Used for function pointers and returns

Pseudo instruction **`ret rs1`** → realized as **`jalr x0, rs1, 0`**

If rs1 is not specified, use x1 (ra)

RISC V Calling Convention

Function Calls (Example)

```
void fact(){
        int f = 1;
        int n = 5;

        while(n > 1){
                f = f * n;
                n = n - 1;
        }
        return;
}

int main(){
        fact();
}
```

```
0000000000010150 <fact>:
   10150:       1101                    addi    sp,sp,-32
   10152:       ec22                    sd      s0,24(sp)

   1018e:       6462                    ld      s0,24(sp)
   10190:       6105                    addi    sp,sp,32
   10192:       8082                    ret
```

Sets PC back to 101a0 (based on the ra register)

PC changed to 10150; ra set to 101a0

```
   1019c:       fb5ff0ef                jal     ra,10150 <fact>
   101a0:       4781                    li      a5,0
```

More Pseudo Instructions

- Example 4: Unconditional Jumps
  Pseudo instruction **`j label`** translates to **`jal x0, label`**

  Pseudo instruction **`jr label`** translates to **`jalr x0, rs1, 0`**

- Example 5: return
  Pseudo instruction **`ret`** translates to **`jalr x0, ra, 0`**

RISC V Assembly Program Manual: https://shakti.org.in/docs/risc-v-asm-manual.pdf
RISC V Assembly Programmer's Manual: https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md
RISC V Programming Book: https://riscv-programming.org/book/riscv-book.html

Multiple function invocations

```
int dec(int n){
  return n--;
}

void fact(){
  int f = 1;
  int n = 5;

  while(n > 1){
    f = f * n;
    n = dec(n);
  }
  return;
}

int main(){
  fact();
}
```

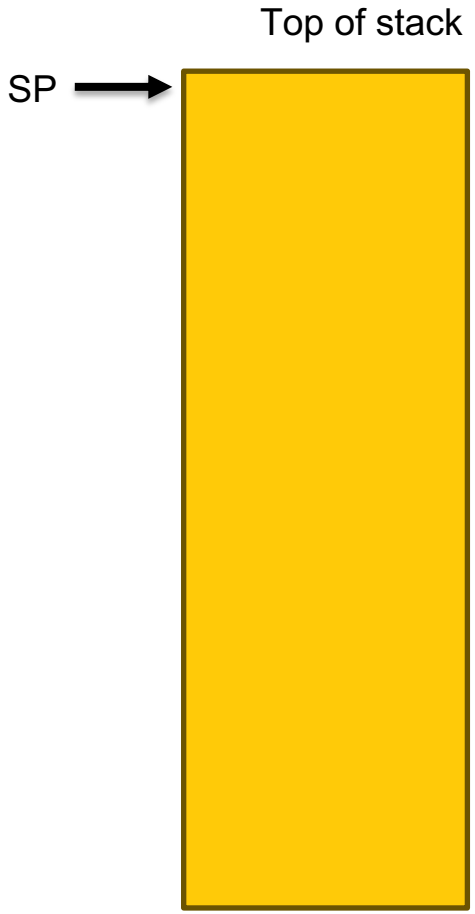
The jal here would over wite the ra register

Won’t return to main (as ra is overwritten)

The jal here store PC+4 in the ra register

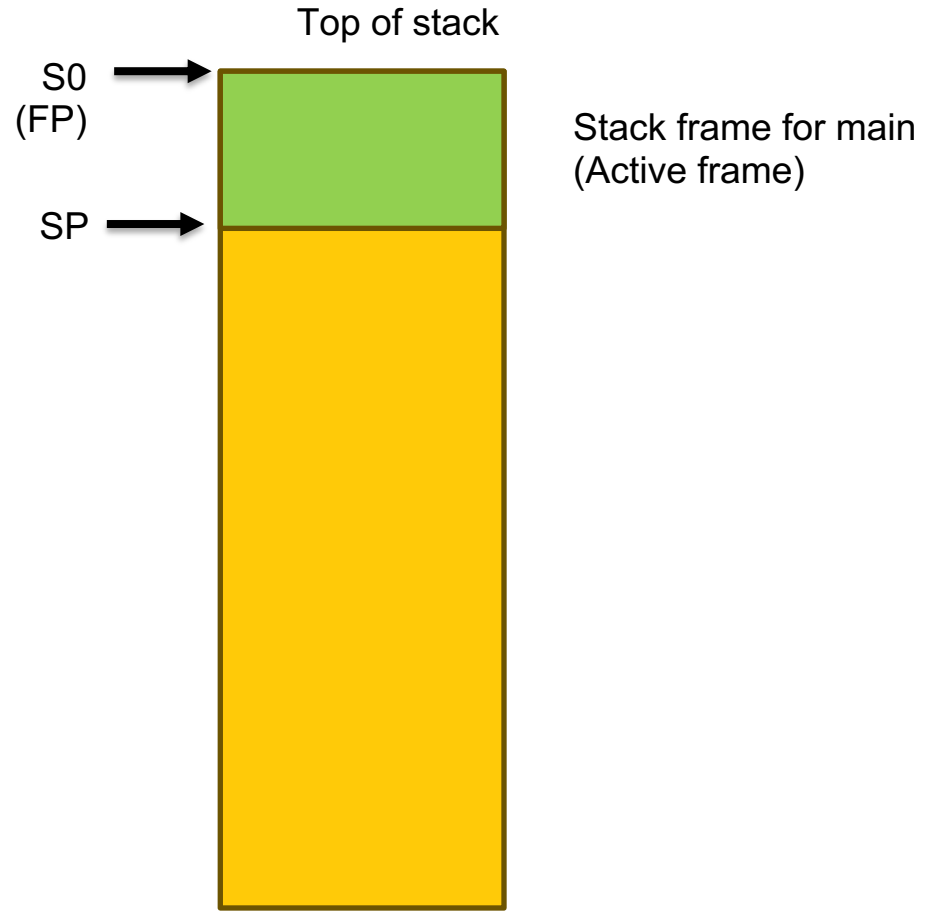
The Stack

- Used to store return addresses
- Also used to store local variables
- Grows from high address to low address
- Register sp (x2) used to point to the lowest address in the stack



Functioning of Stack

Stack grows with each function call.




```
int main(){
  fact();
}
```

Corresponding to each function that is executing, there is a stack frame defined between the registers S0 (also called frame pointer (FP)) and stack pointer SP

Functioning of Stack

Stack grows with each function call.

```
void fact(){
  int f = 1;
  int n = 5;

  while(n > 1){
    f = f * n;
    n = dec(n);
  }
  return;
}


int main(){
  fact();
}
```

Prologue: At the start of fact function

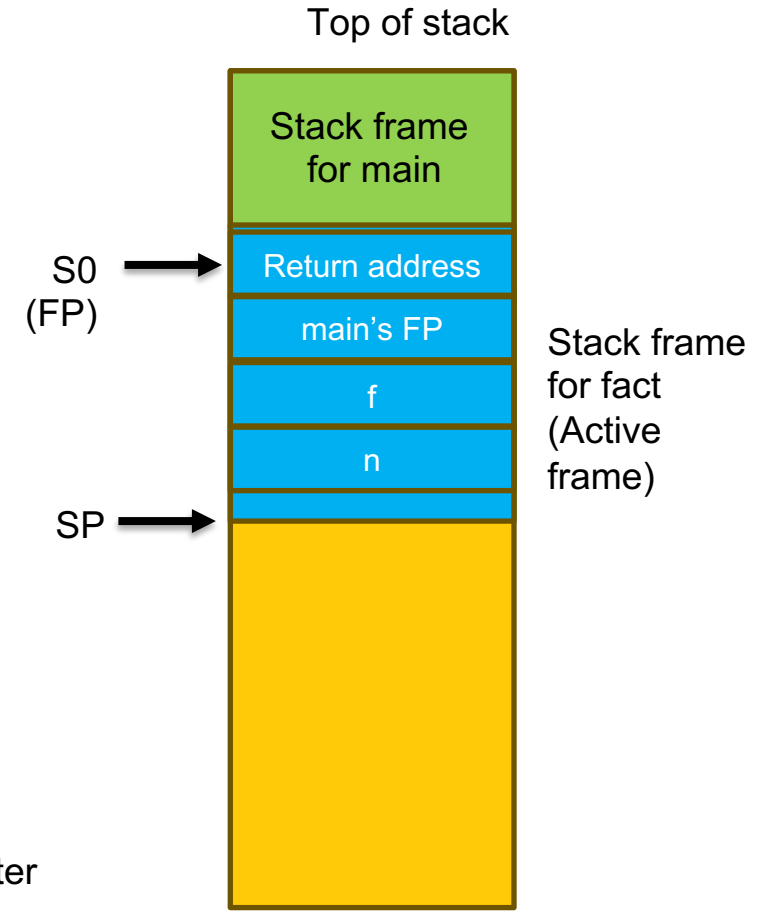
```
addi    sp,sp,-32   Create a stack frame (32 bytes)
sd      ra,24(sp)   Store the return address in stack
sd      s0,16(sp)   Store S0 in stack
addi    s0,sp,32    S0 points to start of stack frame
```

Create the local variables

```
li      a5,1
sw      a5,-20(s0)  f
li      a5,5
sw      a5,-24(s0)  n
```

Access local variables

```
lw      a4,-20(s0)
lw      a5,-24(s0)
mulw    a5,a4,a5
```



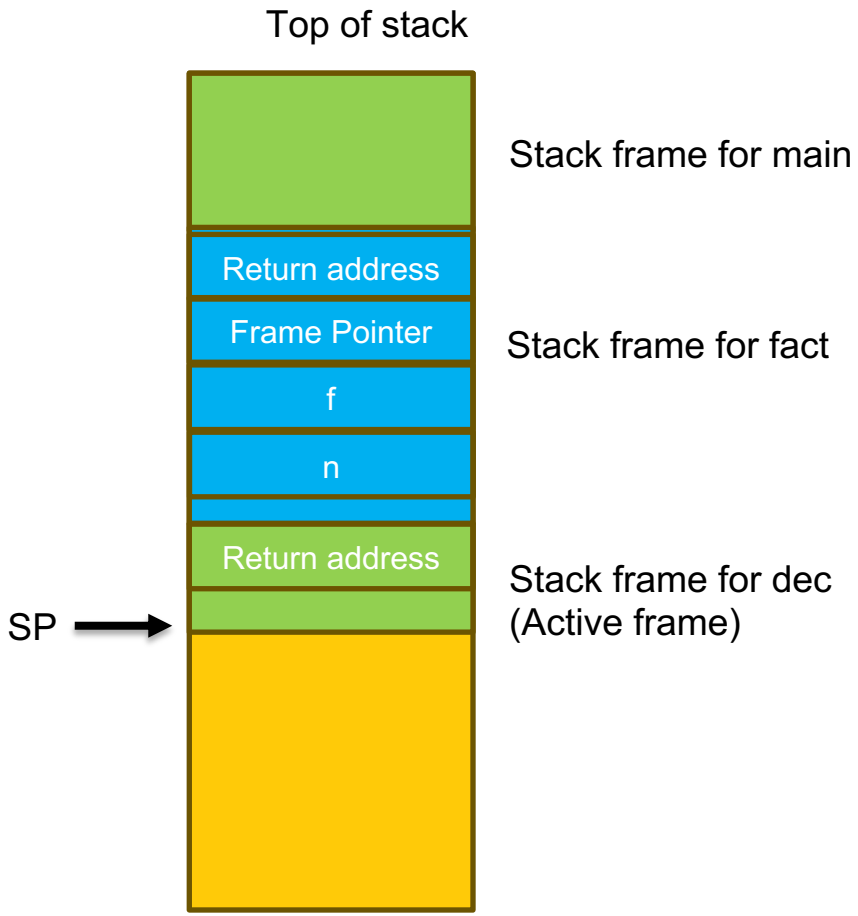
s0 (aka FP): Frame pointer
sp : stack pointer

The stack frame of a function is used to store the previous frame pointer, return address local variables, and any temporary data that needs to be saved.

Functioning of Stack

```
int dec(int n){
  return n--;
}

void fact(){
  int f = 1;
  int n = 5;

  while(n > 1){
    f = f * n;
    n = dec(n);
  }
  return;
}


int main(){
  fact();
}
```

Stack grows with each function call.



Functioning of Stack

Stack shrinks with each return

```
void fact(){
  int f = 1;
  int n = 5;

  while(n > 1){
    f = f * n;
    n = dec(n);
  }
  return;
}


int main(){
  fact();
}
```

Prologue: At the start of fact function

```
addi    sp,sp,-32   Create a stack frame (32 bytes)
sd      ra,24(sp)   Store S0 in stack
sd      s0,16(sp)   Store the return address in stack
addi    s0,sp,32    S0 points to start of stack frame
```

Create the local variables

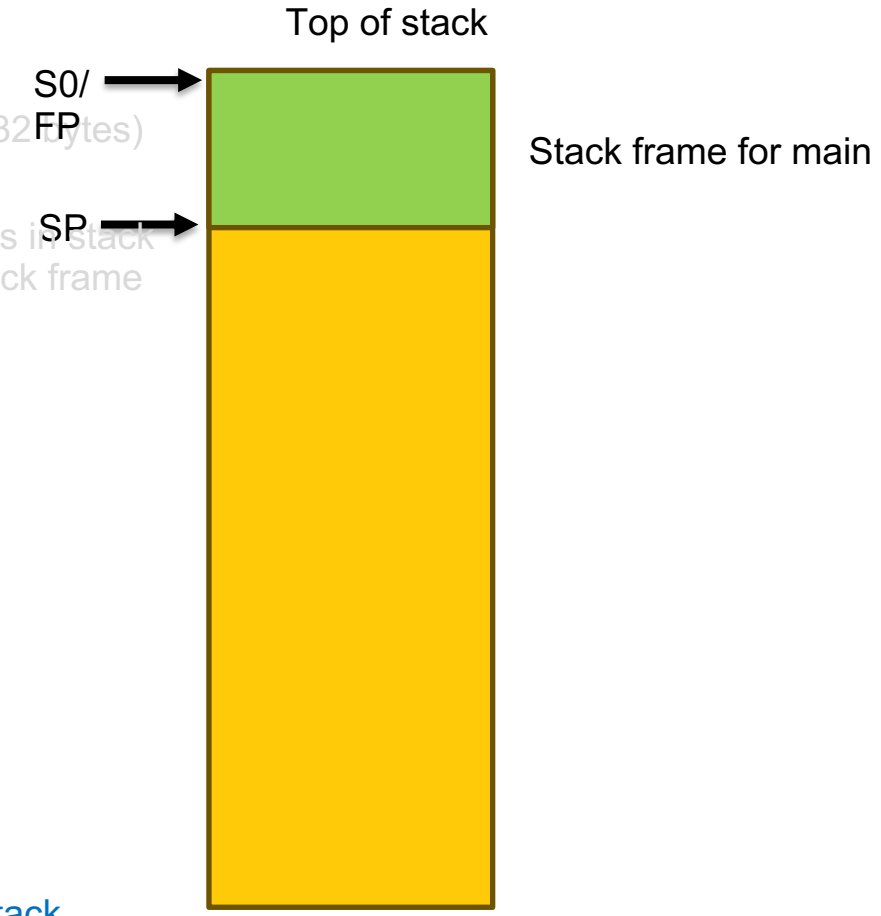
```
li      a5,1
sw      a5,-20(s0)  f
li      a5,5
sw      a5,-24(s0)  n
```

Create the local variables

```
lw      a4,-20(s0)
lw      a5,-24(s0)
mulw    a5,a4,a5
```

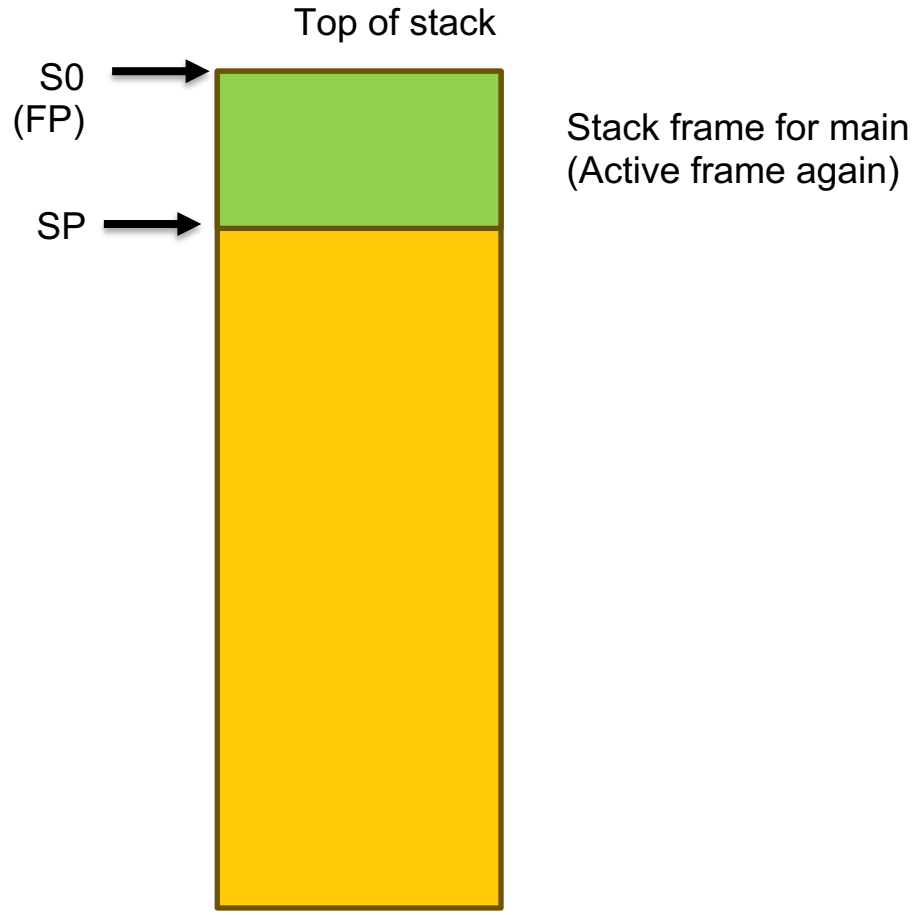
Epilogue

```
ld      ra,24(sp)   Restore RA from the stack
ld      s0,16(sp)   Restore S0 from the stack
addi    sp,sp,32    Shrink the stack → destroy stack frame
ret                 Return based on register RA
```



Functioning of Stack

Stack shrinks with each return




```
int main(){
  fact();
}
```

Passing parameters to functions

- Parameters passed through registers

x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller

```
int sumup(int a, int b, int c, int d, int e,
          int f, int g, int h, int i, int j){
  int res;

  res = a + b + c + d + e + f + g + h + i + j;
  return res;
}
```

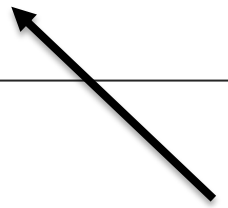
- a, b, c, d, e, f, g, h → passed through registers a0 to a7
- i, j → passed through the stack

```
sumup(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Return data from function

- Return through a0 register

```
int sumup(int a, int b, int c, int d, int e,
          int f, int g, int h, int i, int j){
  int res;

  res = a + b + c + d + e + f + g + h + i + j;
  return res;
}
```



Result returned through a0 register

Example of Parameter Passing and Return

```
int main() {
    int a0 = 5;
    int a1 = 7;

    int result = sum_function(a0, a1);

    return 0;
}

int sum_function(int a, int b) {
    return a + b;
}
```

```
.section .text
.global _start

_start:
    # Example parameters
    li  a0, 5         # First parameter
    li  a1, 7         # Second parameter

    jal ra, sum_function

    ret

.global sum_function
sum_function:
    # Sum the parameters a0 and a1
    add  a0, a0, a1

    ret
```

The two parameters a0 and a1 are passed through resp. registers
The return from sum_function is in the a0 register

Caller and Callee Saved Registers

Numbers hardware understands

Human-friendly symbolic names in assembly code

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	t0	Temporary/alternate link register	Caller
x6–7	t1–2	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

Caller Saved (volatile / call-clobbered)

- Holds temporary data used in functions
- Caller function's responsibility to push these data onto the stack before calling any function
- These values will be restored when the function returns

```
someFunction(){
   .
   .
   someOtherFunction();
   .
   .
}
```

Save all the caller saved registers that you use in this function

Restore caller saved registers here (only if needed)

Caller and Callee Saved Registers

Numbers hardware understands

Human-friendly symbolic names in assembly code

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	t0	Temporary/alternate link register	Caller
x6–7	t1–2	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

Callee Saved (non-volatile / call-preserved)

- Holds temporary data used in functions
- Callee function's responsibility to push these data onto the stack at the start of the function
- These values will be restored when the function ends

```
someOtherFunction(){
    .
    .
    .
    .
    .
}
```

Save all the callee saved registers on stack that you use in this function

Restore callee saved registers here

Example for Caller vs Callee Saved Registers

```
.global caller_saved_demo

caller_saved_demo:

    # Function body
    add  t0, a0, a1
    add  t1, t0, a2


    ret
```

Callee saved registers like t0 and t1 do not need to be saved and restored in the function.

```
.global callee_saved_demo

callee_saved_demo:
    # Save callee-saved registers
    add  sp, sp, -4
    sw   s0, 0(sp)
    add  sp, sp, -4
    sw   s1, 0(sp)

    # Function body
    add  s0, a0, a1
    add  s1, s0, a2

    # The result is stored in s1

    # Restore callee-saved registers
    lw   s1, 0(sp)
    add  sp, sp, 4
    lw   s0, 0(sp)
    add  sp, sp, 4

    ret
```

Caller saved registers like s0 and s1 saved and restored in the function before and after use Resp.

More examples of Caller saved register

```
caller_saved_demo:

    # Function body
    add  t0, a0, a1

    ret
```

The function modifies t0 register, which is a callee saved register

```
_start:
    # Initialize arguments
    li  a0, 5
    li  a1, 7
    li  a2, 10

    # Call the caller-saved function
    jal ra, caller_saved_demo

    ret
```

At the caller, no need to save t0, because it is not used in the function

```
_start:
    # Use t0 for some computation
    li   t0, 10
    add  t0, t0, a0

    # Save the value of t0 onto the stack
    addi sp, sp, -4
    sw   t0, 0(sp)

    # Call the callee function
    jal ra, caller_function_demo

    # Restore the value of t0 from the stack
    lw   t0, 0(sp)
    addi sp, sp, 4

    # Continue in the caller function if needed
    # ...

    ret
```

At the caller, save t0 on the stack because it is used in the function. Restore after the caller_function_demo is invoked.