



IP PARIS



Building a RISC-V Processor

ECE_3TC31_TP/INF107

Ulrich Kühne Florian Brandner Tarik Graba Guillaume Duc
2023



Introduction

Overview – Putting it all together

In this chapter, you will find answers to the following questions:

- What's inside a computer?
- What does a processor do?
- How to talk to a processor?
- How to build a (simple) processor?

What's inside a computer?

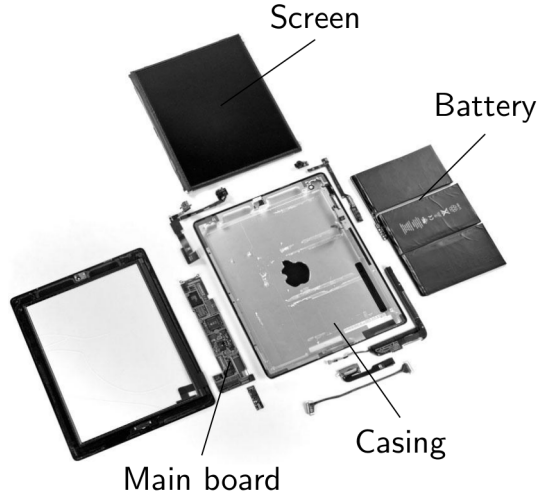


Figure 1: Things you typically find in a computer

The Main Board

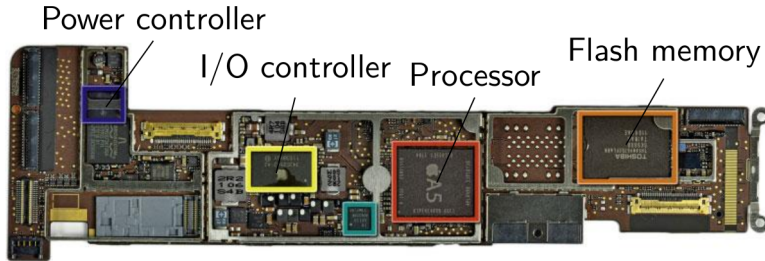
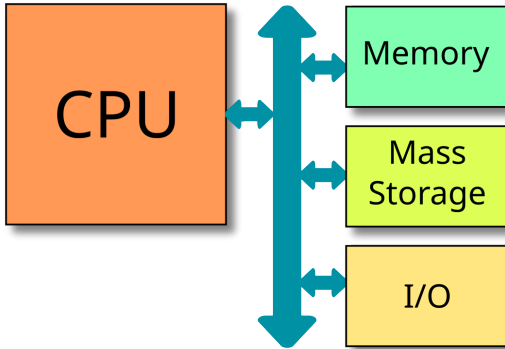


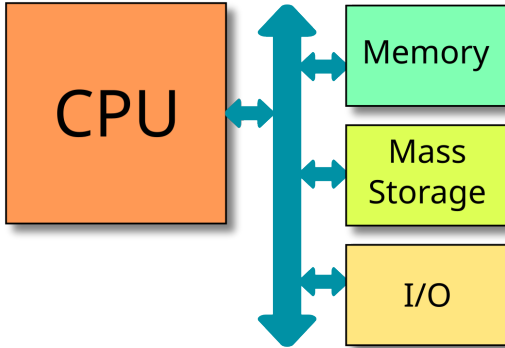
Figure 2: Main circuit board with different components

A More Systematic View on Things



- Central Processing Unit (CPU, processor)
- Memory (random access memory, main memory)
- Mass storage (hard disk, SSD)
- Input/output peripherals
 - Keyboard
 - Mouse or touchpad
 - Screen display
 - Audio input and output
 - Network devices (WiFi, ethernet)
 - Many more...

A Word on Interconnections



- Components are interconnected via busses
- Serial bus
 - Few pins and cables
 - Slow communication
 - Used for external peripherals
 - Examples: USB, I2C, SPI
- Parallel bus
 - Many pins required
 - Fast communication
 - Used for on-board communication
 - Examples: PCI, AXI
- Wireless communication
 - Examples: Bluetooth, WiFi, ZigBee

The Central Processing Unit



- Performs (most of) the computations
- Reads data from memory or peripherals
- Processes it
- Sends result back to memory or peripherals

Processor Architecture

When we talk about **processor architecture**, this can mean different things

Instruction Set Architecture (ISA)

- Determines the elementary operations (instructions) a processor can perform
- N -bit architecture ($N \in \{8, 16, 32, 64, \dots\}$) refers to “natural” size of data that is processed (register size, size of busses, ...)

Micro-architecture

- Refers to the internal organisation of a specific processor or a family of processors in order to implement its ISA
- The same ISA can be realized in many different ways

Some Instruction Set Architectures

■ ARM

- Family of ISAs developed by ARM
- Used in embedded systems (mobile and low power) and desktop (Apple's M1)

■ x86

- Family of ISAs developed by Intel (and AMD)
- Used in general purpose computing systems (desktop and servers)

■ RISC-V

- Family of open standard ISAs developed by University of California, Berkeley
- Mostly used in embedded systems

■ MOS 6502

- Historical 8 bit architecture
- Used in first home computers (Commodore 64, Apple II) and game consoles (Atari)



Performance and Trade-Offs

Performance indicators of a processor

- Instructions executed per second
- Logic complexity (e.g. number of gates)
- Power consumption
- Predictability / real-time behavior
- ...

Different architectural trade-offs

- | | |
|---|---|
| <ul style="list-style-type: none">■ General purpose processor (GPP)<ul style="list-style-type: none">• High average performance• High cost and power consumption• Poor predictability | <ul style="list-style-type: none">■ Embedded micro-controller<ul style="list-style-type: none">• Low performance• Low cost and power consumption• Good predictability |
|---|---|

The Machine Language

How to Talk to a Processor?

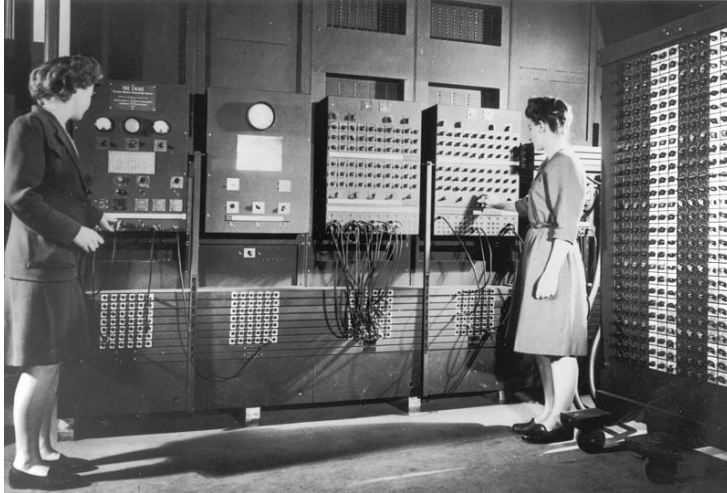
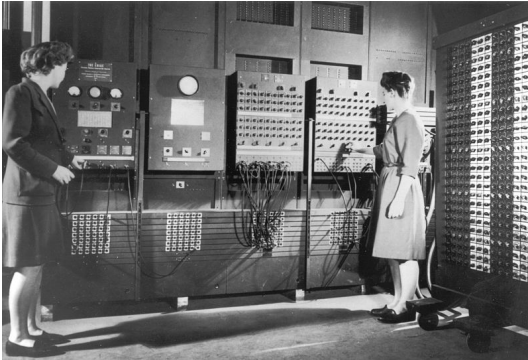


Figure 3: Betty Jean Jennings and Fran Bilas programming the ENIAC

How to Talk to a Processor?



- Programming language
 - Symbolic
 - Human readable (hopefully)
 - Use of variables and functions
- Machine language
 - Concrete low level instructions
 - Machine readable
 - Binary (ones and zeros)

How to close this gap?

From High Level Language to Machine Language

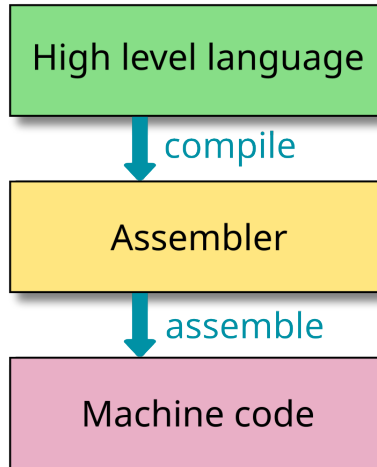


Figure 4: The compilation process

Compilation

Input:

```
int square(int x) {  
    return x * x;  
}  
  
int main() {  
    int x = square(42);  
}
```

Output:

```
square:  
    addi    sp,sp,-16  
    sw      ra,12(sp)  
    sw      s0,8(sp)  
    addi    s0,sp,16  
    sw      a0,-16(s0)  
    lw      a1,-16(s0)  
    lw      a0,-16(s0)  
    call    __mulsi3  
    mv      a5,a0  
    mv      a0,a5  
    lw      ra,12(sp)  
    lw      s0,8(sp)  
    addi    sp,sp,16  
    jr      ra  
    .align  2  
main:  
    addi    sp,sp,-16  
    sw      ra,12(sp)  
    sw      s0,8(sp)  
    addi    s0,sp,16  
    li      a0,42  
    call    square  
    sw      a0,-16(s0)  
    li      a5,0  
    mv      a0,a5  
    lw      ra,12(sp)  
    lw      s0,8(sp)  
    addi    sp,sp,16  
    jr      ra
```


Assembler Code

```
square:
    addi    sp,sp,-16
    sw      ra,12(sp)
    sw      s0,8(sp)
    addi    s0,sp,16
    sw      a0,-16(s0)
    lw      a1,-16(s0)
    lw      a0,-16(s0)
    call    __mulsi3
    mv      a5,a0
    mv      a0,a5
    lw      ra,12(sp)
    lw      s0,8(sp)
    addi    sp,sp,16
    jr      ra
    .align  2
main:
    addi    sp,sp,-16
    sw      ra,12(sp)
    sw      s0,8(sp)
    addi    s0,sp,16
    li      a0,42
    call    square
    sw      a0,-16(s0)
    li      a5,0
    mv      a0,a5
    lw      ra,12(sp)
    lw      s0,8(sp)
    addi    sp,sp,16
    jr      ra
```

- ISA-specific
- Textual representation of
 - Symbolic labels
 - Instructions
 - Registers
 - Literals
- Further directives (e.g. alignment)
- Fixed number of registers
- No complex control flow (no loops)
- Lowest human-readable level

Assembler

Input:

```
square:
    addi sp,sp,-16
    sw   ra,12(sp)
    sw   s0,8(sp)
    addi s0,sp,16
    sw   a0,-16(s0)
    lw   a1,-16(s0)
    lw   a0,-16(s0)
    call __mulsi3
    mv   a5,a0
    mv   a0,a5
    lw   ra,12(sp)
    lw   s0,8(sp)
    addi sp,sp,16
    jr   ra
    .align 2
main:
    addi sp,sp,-16
    sw   ra,12(sp)
    sw   s0,8(sp)
    addi s0,sp,16
    li   a0,42
    call square
    sw   a0,-16(s0)
    li   a5,0
    mv   a0,a5
    lw   ra,12(sp)
    lw   s0,8(sp)
    addi sp,sp,16
    jr   ra
```

Output:

00000000	0113	ff01	2623	0011	2423	0081	0413	0101
00000010	2823	fea4	2583	ff04	2503	ff04	0097	0000
00000020	80e7	0000	0793	0005	8513	0007	2083	00c1
00000030	2403	0081	0113	0101	8067	0000	0113	ff01
00000040	2623	0011	2423	0081	0413	0101	0513	02a0
00000050	0097	0000	80e7	0000	2823	fea4	0793	0000
00000060	8513	0007	2083	00c1	2403	0081	0113	0101
00000070	8067	0000						
00000074								

Instruction Classes (in RISC-V)

```
square:
  addi sp,sp,-16
  sw   ra,12(sp)
  sw   s0,8(sp)
  addi s0,sp,16
  sw   a0,-16(s0)
  lw   a1,-16(s0)
  lw   a0,-16(s0)
  call __mulsi3
  mv   a5,a0
  mv   a0,a5
  lw   ra,12(sp)
  lw   s0,8(sp)
  addi sp,sp,16
  jr   ra
  .align 2
main:
  addi sp,sp,-16
  sw   ra,12(sp)
  sw   s0,8(sp)
  addi s0,sp,16
  li   a0,42
  call square
  sw   a0,-16(s0)
  li   a5,0
  mv   a0,a5
  lw   ra,12(sp)
  lw   s0,8(sp)
  addi sp,sp,16
  jr   ra
```

Arithmetic and logic instructions

- Addition, subtraction
- Bitwise and
- Left shift
- ...

Memory access instructions

- Load (fetch data from memory to a register)
- Store (save data from a register to memory)

Control flow

- Unconditional jump
- Conditional branch
- Call (save return address before jumping)

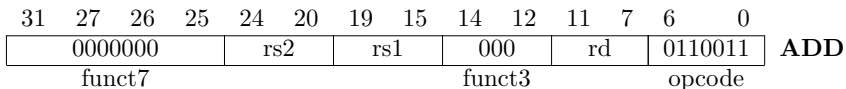
How to execute an instruction?

1. Read the instruction word from memory (**fetch**)
2. Determine the type of the instruction and its operands (**decode**)
3. Perform the demanded computations (**execute**)
4. Store the result in the requested location (**write back**)
5. Determine the **next instruction** and start again

The RISC-V Instruction Set

- Fixed size of 32 bits
- 32 registers
- Load-store architecture
 - Arithmetic and logic instructions working on registers only
 - Specific instructions to move data from and to memory
- Reduced Instruction Set (RISC) ISA
 - Few (49) instructions in the base ISA
 - Few and regular instruction formats
 - Allows for very small hardware implementations
- Standardized extensions of base ISA
 - Multiplication and division
 - Floating point
 - Bit manipulation
 - Vector computations
 - ...

Example of A RISC-V Instruction

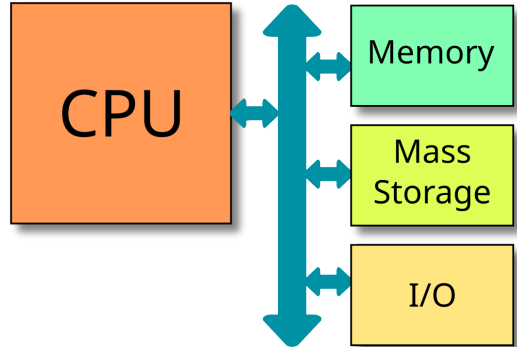


- *What to do?*
 - opcode (register-to-register instruction)
 - function fields (addition)
- *Where to get the operands?*
 - opcode (register-to-register instruction)
 - rs1/rs2 (source registers)
- *Where to put the result?*
 - rd (destination register)

Building a RISC-V Processor

Basic Ingredients

- Memory
- Processor
 - Registers
 - Lots of logic...
- We will ignore peripherals for now



Memory

- Separate instruction and data memory (*Harvard architecture*)
- Combinatorial read (result in the same cycle)
- Synchronous write (update at rising edge)

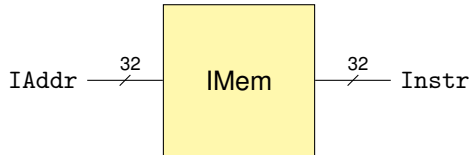


Figure 5: Instruction memory

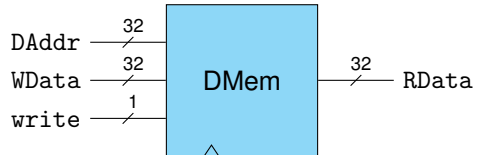
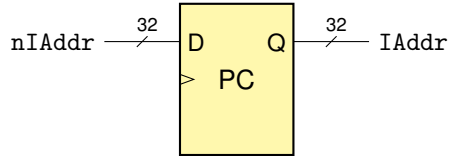


Figure 6: Data memory

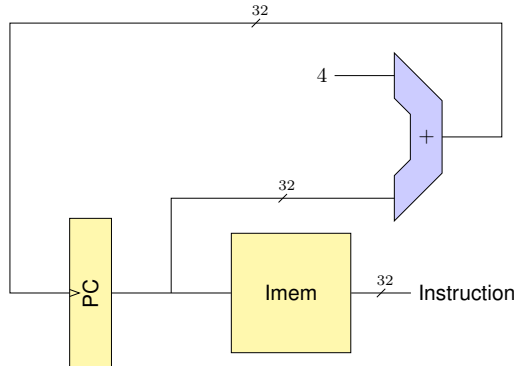
Program Counter

- 32 bit Register
- Stores address of current instruction
- Initialised to address 0



Putting things together: Fetching instructions

1. **Read the instruction word from memory**
 2. Decode the instruction
 3. Perform the demanded computations
 4. Store the result
 5. **Determine the next instruction**
- Program starts at address 0 (simplification)
 - Fixed size instructions of 32 bits (4 bytes)
 - Fetch consecutive instructions



Doing the actual work

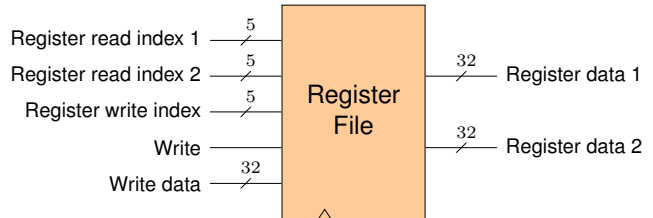
1. Read the instruction word from memory
2. **Decode the instruction**
3. **Perform the demanded computations**
4. **Store the result**
5. Determine the next instruction

What do we need?

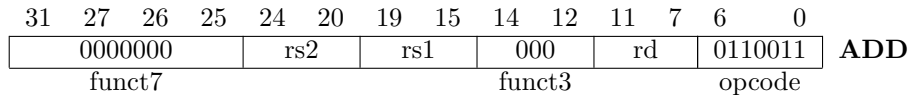
- A circuit to decode the instruction
- A place to store operands and results
- Arithmetic and logic operators

Register File

- 32 registers of 32 bit
 - Register 0 hard wired to 0
- Two separate read ports
- One write port
- Combinatorial read
- Synchronous write



Decoding instructions



- *What to do?*
 - opcode (register-to-register instruction)
 - function fields (addition)
- *Where to get the operands?*
 - opcode (register-to-register instruction)
 - rs1/rs2 (source registers)
- *Where to put the result?*
 - rd (destination register)

R-type Instructions

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
0000000				rs2		rs1		000		rd		0110011		ADD
0100000				rs2		rs1		000		rd		0110011		SUB
0000000				rs2		rs1		001		rd		0110011		SLL
0000000				rs2		rs1		010		rd		0110011		SLT
0000000				rs2		rs1		011		rd		0110011		SLTU
0000000				rs2		rs1		100		rd		0110011		XOR
0000000				rs2		rs1		101		rd		0110011		SRL
0000000				rs2		rs1		110		rd		0110011		OR
0000000				rs2		rs1		111		rd		0110011		AND

Figure 7: R-type instructions

Exercise: Encoding and Decoding Instructions

Encoding

Encode the following instructions¹:

```
xor x8, x4, x5  
sub x6, x15, x1
```

Decoding

Decode the following instructions²:

```
0x003110b3  
0x007067b3
```

¹In the assembler notation, the order of the registers is **rd**, **rs1**, **rs2**. The notation **xi** refers to the register with index *i*

²The prefix **0x** is a common notation for numbers in hexadecimal format

Solution: Encoding and Decoding Instructions

Encoding

Encode the following instructions:

```
xor x8, x4, x5    # 0x00524433 (0000 0000 0101 0010 0100 0100 0011 0011)
sub x6, x15, x1    # 0x40178333 (0100 0000 0001 0111 1000 0011 0011 0011)
```

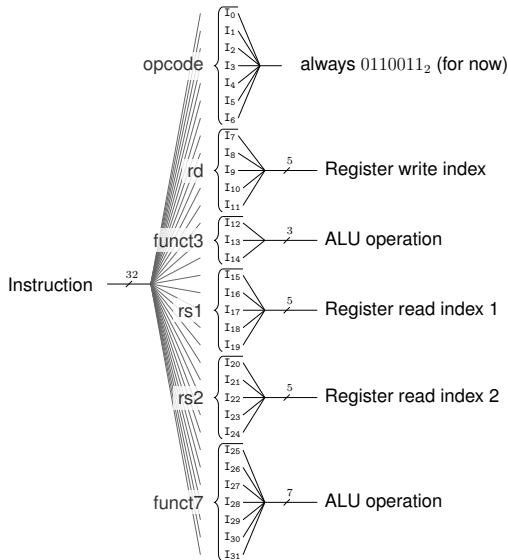
Decoding

Decode the following instructions³:

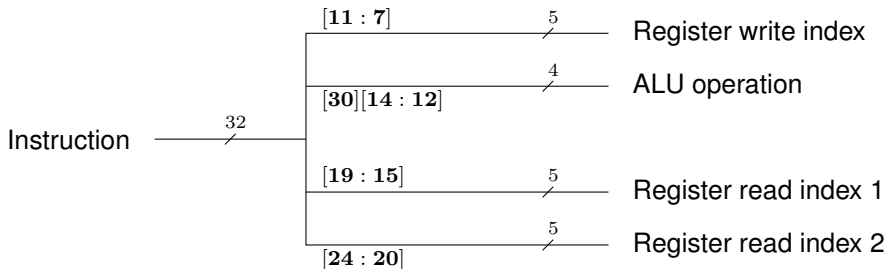
```
0x003110b3 # sll  x1, x2, x3
0x007067b3 # or   x15, x0, x7
```

³You can find an online encoder/decoder here: <https://luplab.gitlab.io/rvcodecs/>

Decode Unit for R-type Instructions

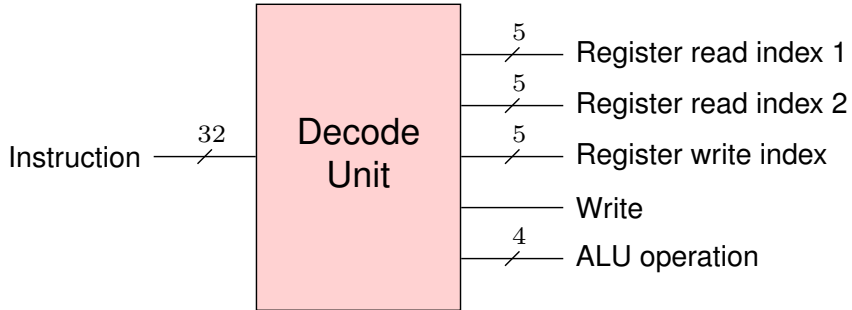


Decode Unit for R-type Instructions

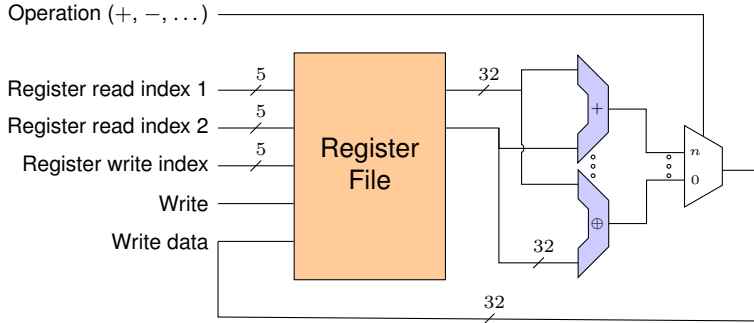


- Decode by redirecting wires (for now)
- Only need bit 6 of *funct7* (bit 30 of instruction)

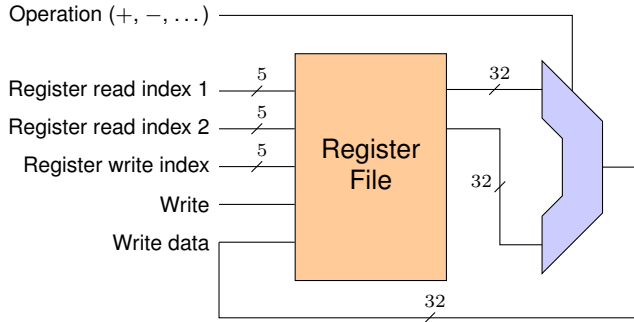
Decode Unit for R-type Instructions



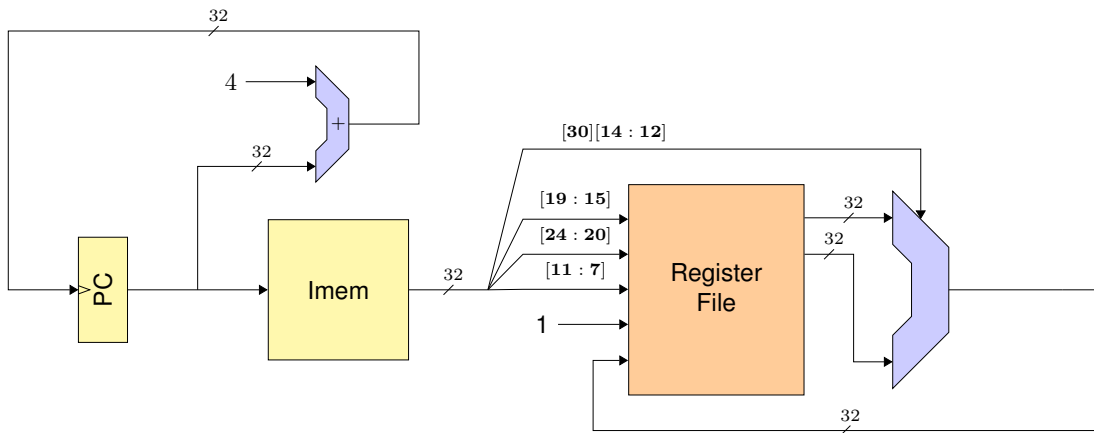
Putting things together: R-type Instructions



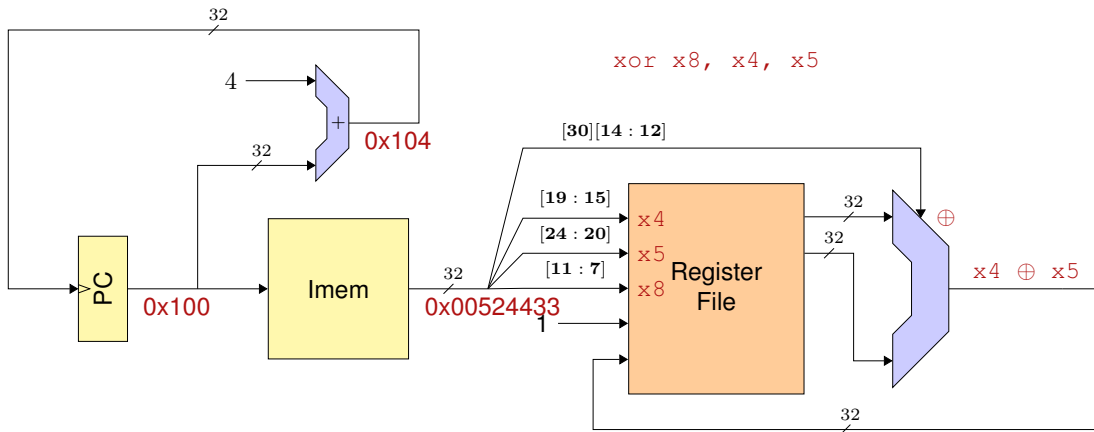
Putting things together: R-type Instructions



Putting things together: R-type Instructions



Putting things together: R-type Instructions



Going Further

- First working processor implementation
- How to use constant values
 - in computations?
 - to initialise registers?



Immediate Instructions

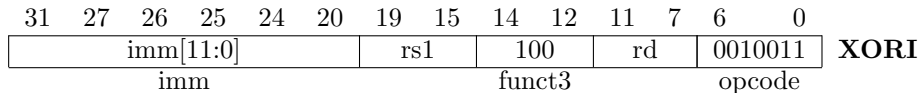


Figure 8: Immediate instruction `xori`

- New opcode `0010011`
- Constant encoded in instruction word
- 12 bit integer value $\in [-2048, 2047]$
- Sign extension to 32 bits
- Use register `rs1` as second operand
- Store result in register `rd`

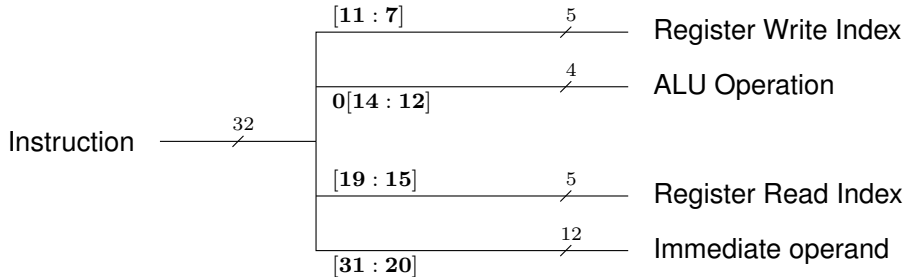
Immediate Instructions

31	27	26	25	24	20	19	15	14	12	11	7	6	0
imm[11:0]						rs1	funct3		rd		opcode		

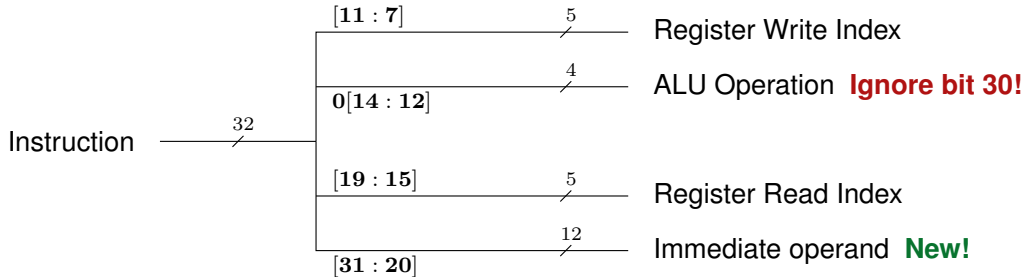
I-type

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
imm[11:0]						rs1	000		rd		0010011			ADDI
imm[11:0]						rs1	010		rd		0010011			SLTI
imm[11:0]						rs1	011		rd		0010011			SLTIU
imm[11:0]						rs1	100		rd		0010011			XORI
imm[11:0]						rs1	110		rd		0010011			ORI
imm[11:0]						rs1	111		rd		0010011			ANDI
0000000				shamt		rs1	001		rd		0010011			SLLI
0000000				shamt		rs1	101		rd		0010011			SRLI

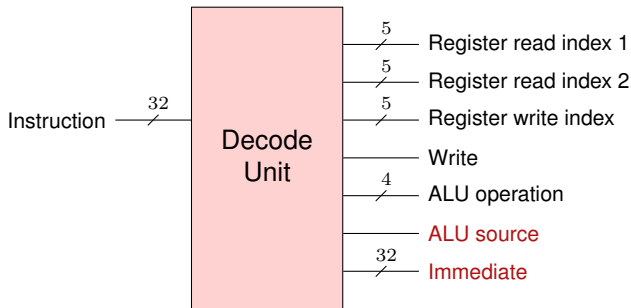
Decoding Immediate Instructions



Decoding Immediate Instructions

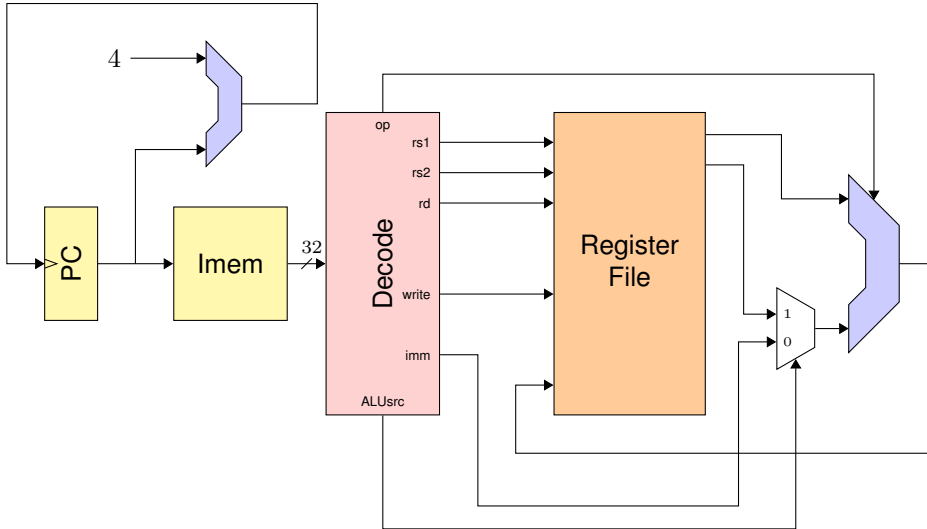


Decoder Unit for R-type and I-type Instructions



- Sign extended *Immediate* value
- ALU source signal to select between *rs2* (1) and *Immediate* (0)
- Implementation left as an exercise

Putting things together: R-type and I-type Data Path



What's missing?

- Why is there no `subi` (*subtract immediate*) instruction?
- Why is there no `not` (bitwise negation) instruction?

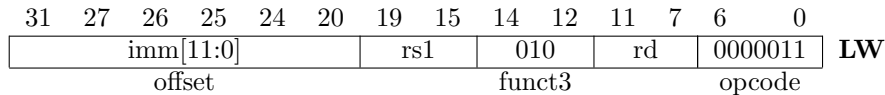


What else?

- How to use data from memory?
- How to store results in memory?

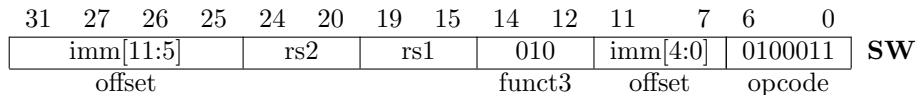


Load Instruction(s)



- Load word (32 bits) from memory to register *rd*
- Address from register *rs1* plus immediate offset
- Similar instructions for different data sizes
 - **lb**: load byte (8 bits)
 - **lh**: load half-word (16 bits)
- Assembler syntax: **lw** *s0*, 8(*a0*) loads $\text{MEM}[a0 + 8]$ in register *s0*

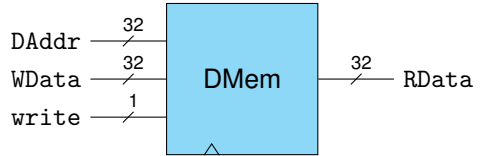
Store Instruction(s)



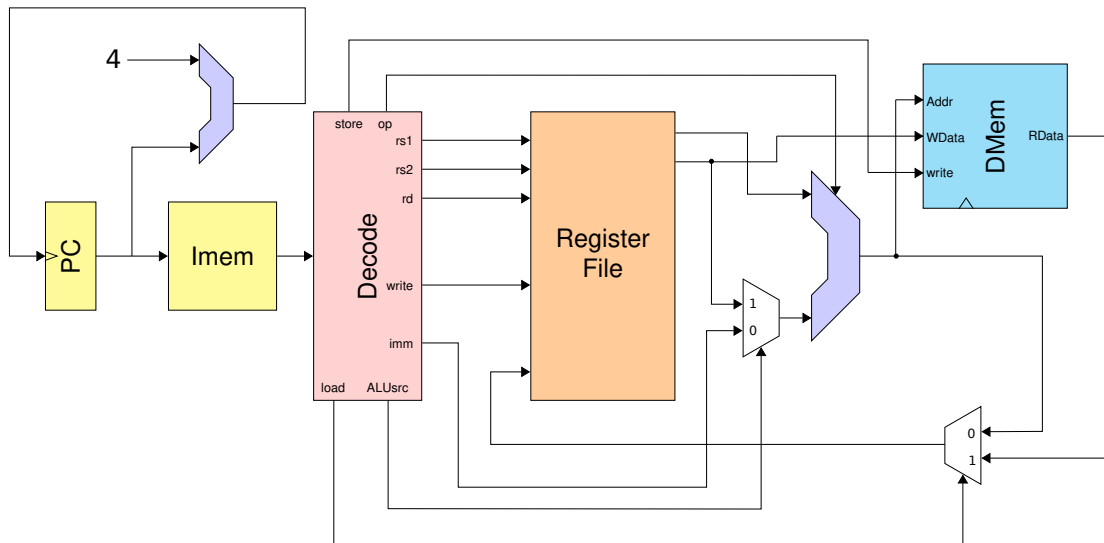
- Store word (32 bits) to memory from register *rs2*
- Address from register *rs1* plus immediate offset
- Similar instructions for different data sizes
 - **sb**: store byte (8 bits)
 - **sh**: store half-word (16 bits)
- Assembler syntax: **sw** *s0*, 8(**a0**) stores register *s0* in $\text{MEM}[a0 + 8]$

Data Path for Load/Store Instructions

Propose a data path for load and store!

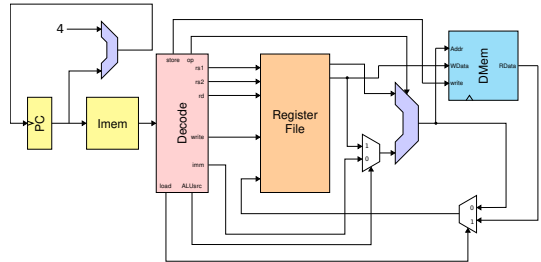


Data Path for Load/Store Instructions



Data Path for Load/Store Instructions

- New control signals *load* and *store*
- New multiplexer to choose data to write back:
 - ALU result (*load* = 0)
 - Memory read data (*load* = 1)
- Use ALU to compute memory address offset
 - ALU operation is addition
 - Select immediate input (*ALUsrc* = 0)
- Register *rs2* fed to memory data input

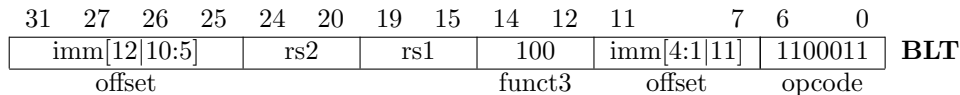


Adding Control Flow

- How to jump to another address?
- How to jump depending on a certain condition?
- How to implement loops?
- How to call a function?
- How to return from a function?



Conditional Branches



- Instruction **blt**: *Branch if less than*
- New opcode **1100011**
- Tests if register *rs1* is less than register *rs2*
- Jumps to address $PC + \text{offset}$ if condition is true
- Offset in range $[-4096, 4095]$ (LSB is always zero)

Conditional Branches

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
imm[12 10:5]				rs2			rs1		funct3		imm[4:1 11]		opcode		B-type

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
imm[12 10:5]				rs2			rs1			000		imm[4:1 11]		1100011	BEQ BNE BLT BGE BLTU BGEU
imm[12 10:5]				rs2			rs1			001		imm[4:1 11]		1100011	
imm[12 10:5]				rs2			rs1			100		imm[4:1 11]		1100011	
imm[12 10:5]				rs2			rs1			101		imm[4:1 11]		1100011	
imm[12 10:5]				rs2			rs1			110		imm[4:1 11]		1100011	
imm[12 10:5]				rs2			rs1			111		imm[4:1 11]		1100011	

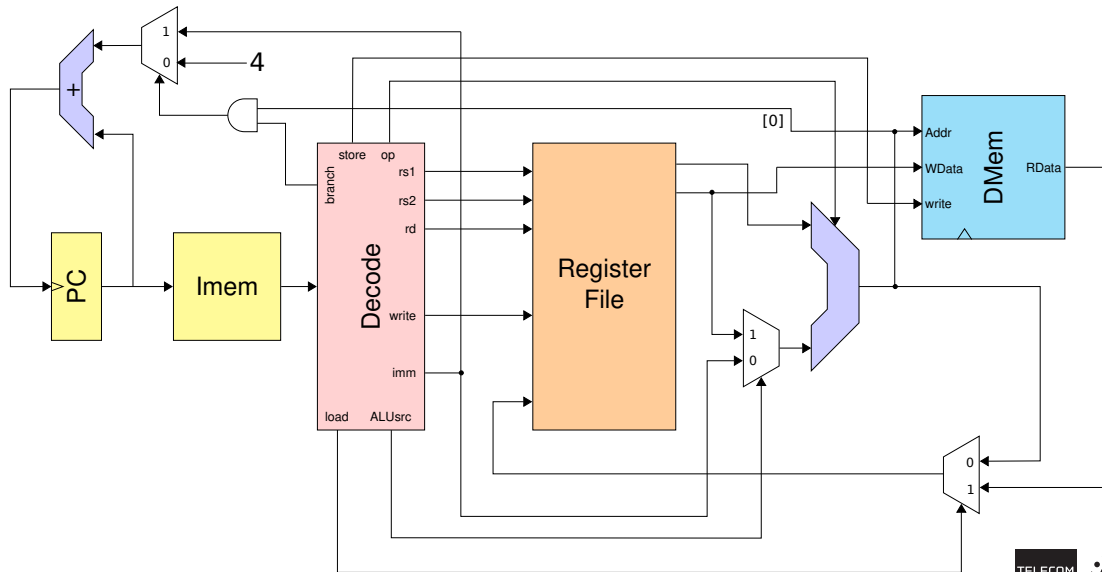
- **beq**: Branch if equal
- **bne**: Branch if not equal
- **blt**: Branch if less than
- **bge**: Branch if greater or equal
- **bltu**: Branch if less than unsigned
- **bgeu**: Branch if greater or equal unsigned



Conditional Branches

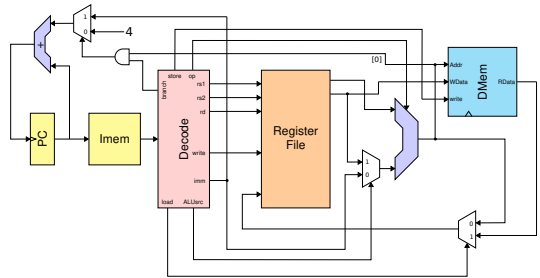
Propose a data path to implement branches!

Data Path for B-type Instructions

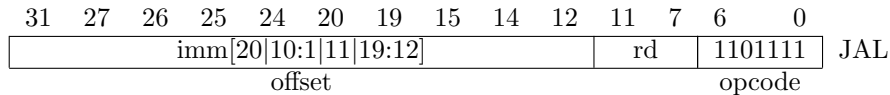


Data Path for B-type Instructions

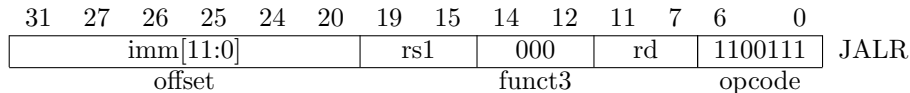
- New control signal *branch*
- Use lowest bit of ALU result for *condition*
- Choose *ALU operation* according to branch condition
- Multiplexer to select *PC* offset (4 or *immediate*)



Unconditional Jump/Call



- **jal**: *jump and link*
- Jump to address $PC + \text{offset}$
- Offset in range $[-1.048.576, 1.048.575]$
- Store $PC + 4$ in register *rd*



- **jalr**: *jump and link register*
- Same as **jal**, with target address register *rs1* plus *offset*



Data Path for Jump Instructions

What do we need to add in order to implement jumps (not shown in this lecture)?

Miscellaneous Instructions

Some instructions not covered in this lecture:

- **lui**: *load upper immediate* (→ homework)
- **auipc**: *add upper immediate to PC* (used for long jumps)
- **ecall**, **ebreak**: switching privilege level (→ third part of this lecture)

RISC-V Assembler

Register Names

Reg	Name	Usage
x0	zero	Constant zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary register 0
x6	t1	Temporary register 1
x7	t2	Temporary register 2
x8	s0 / fp	Saved register 0 / frame pointer
x9	s1	Saved register 1
x10	a0	Function argument 0 / return value 0
x11	a1	Function argument 1 / return value 1

Reg	Name	Usage
x12	a2	Function argument 2
...
x17	a7	Function argument 7
x18	s2	Saved register 2
...
x27	s11	Saved register 11
x28	t3	Temporary register 3
...
x31	t6	Temporary register 6

Pseudo Instructions

- Convenient names for important use cases
- Leads to more readable code

Mnemonic	Usage	Translated to
<code>nop</code>	No operation	<code>addi zero, zero, 0</code>
<code>mv</code>	Copy register	<code>addi rd, rs, 0</code>
<code>not</code>	Bitwise negation	<code>xori rd, rs, -1</code>
<code>li</code>	Load immediate	<code>(lui +) addi</code>
<code>la</code>	Load address	<code>auipc + addi</code>
<code>j</code>	Jump	<code>jal zero</code>
<code>call</code>	Jump to subroutine	<code>jal ra</code>
<code>ret</code>	Return from subroutine	<code>jalr zero, 0(ra)</code>

Directives

Encoding constant data

- `.byte 0xff`: 8 bit constant value
- `.half 0xeef`: 16 bit constant value
- `.word 0xaabbccdd`: 32 bit constant value
- `.dword 0x00112233aabbccdd`: 64 bit constant value

Alignment

- `.align N` aligns next instruction or data to address divisible by 2^N
- Needed e.g. to align constant data to word boundaries

Labels

- Labels for
 - Jump and branch targets
 - Beginning of functions
 - Location of data
- Use label instead of *constant* jump or branch targets
- Use label to initialise a register with an *address*

```
foo:
    addi t0, t0, 1
    j foo

bar:
    la a0, data
    lw t0, 0(a0)

    .align 2
data:
    .word 0xcafe
```

Example Program

```
foo:
    la s0, data      # Load address of data into s0
    lw a0, 0(s0)     # Load word at data
    lw a1, 4(s0)     # Load word at data + 4
    jal bar          # Call function bar, save return address in ra
    sw a0, 8(s0)     # Store function result at data + 8
    j end            # jump to the end

bar:
    add a0, a0, a1    # Add function arguments, save sum to a0
    ret              # Return to caller site (return address in ra)

    .align 2          # Some data to be processed
data:
    .word 0x0000cafe, 0x00010023, 0x0

end:                 # This is the end
    nop
```

Summary

- Different processor architecture trade-offs
- Programs are compiled to machine code
- Instructions are simple elementary operations
- Example in this lecture: RISC-V base ISA
- Fetch, decode, execute, write-back cycle
- Construct data path from basic logic components
- Low-level programming in assembler