Programming Microprocessor – Instruction Set Architecture III

CS2053 Computer Architecture

Computer Science & Engineering
University of Moratuwa

Sulochana Sooriyaarachchi
Chathuranga Hettiarachchi

Slides adopted from Dr.Dilum Bandara

Encoding Instructions

- Various instruction types
- Limited word size for registers, addresses, and instructions
  - Consider 32bit words in RV32I
  - All the instructions are 32bits
  - Example : If we need to load an immediate value to 32-bit register, how to fit all opcode and operands within 32-bit instruction?
    - Work with small numbers
    - Make compromises

Instruction Formats(Types)

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

- Fields
  - Opcode : 7bits
  - funct 3 : 3 bit function
  - funct 7 : 7 bit function
  - rs1, rs2 : two source registers (5 bits each)
  - rd : destination register (5 bits)

Instruction Formats (6 Types)

- R-Format: instructions using 3 ***register*** inputs
  – `add`, `xor`, `mul` —arithmetic/logical ops
- I-Format: instructions with ***immediates***, loads
  – `addi`, `lw, jalr, slli`
- S-Format: ***store*** instructions: `sw, sb`
  - SB-Format: ***branch*** instructions: `beq, bge`
- U-Format: instructions with ***upper*** immediates
  – `lui`, `auipc` —upper immediate is 20-bits
  - UJ-Format: ***jump*** instructions: `jal`

Instruction Format-Register Type

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions (Register 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

For the complete standard specifications:
https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

Instruction Format-Immediate Type

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions (Immediate 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
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

Instruction Format-Upper Immediate

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

Load Upper Immediate (lui)

lui t1,0x70070

Fill up the upper 20 bits of destination register with immediate value

Add Upper Immediate value and Program Counter (auipc)

auipc a0,0x2

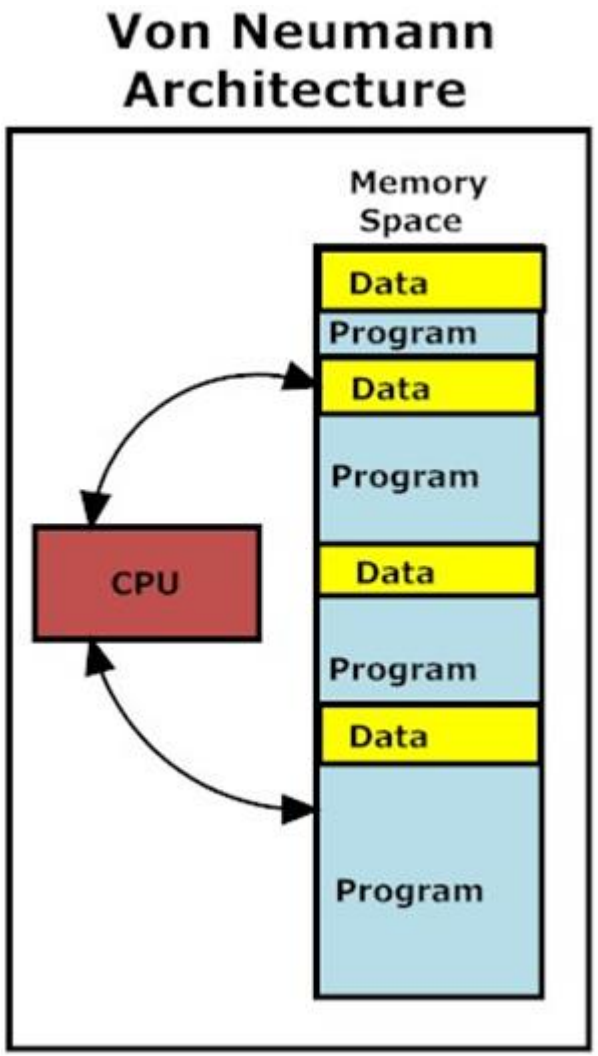
Fill the upper 20 bits of destination register with immediate value

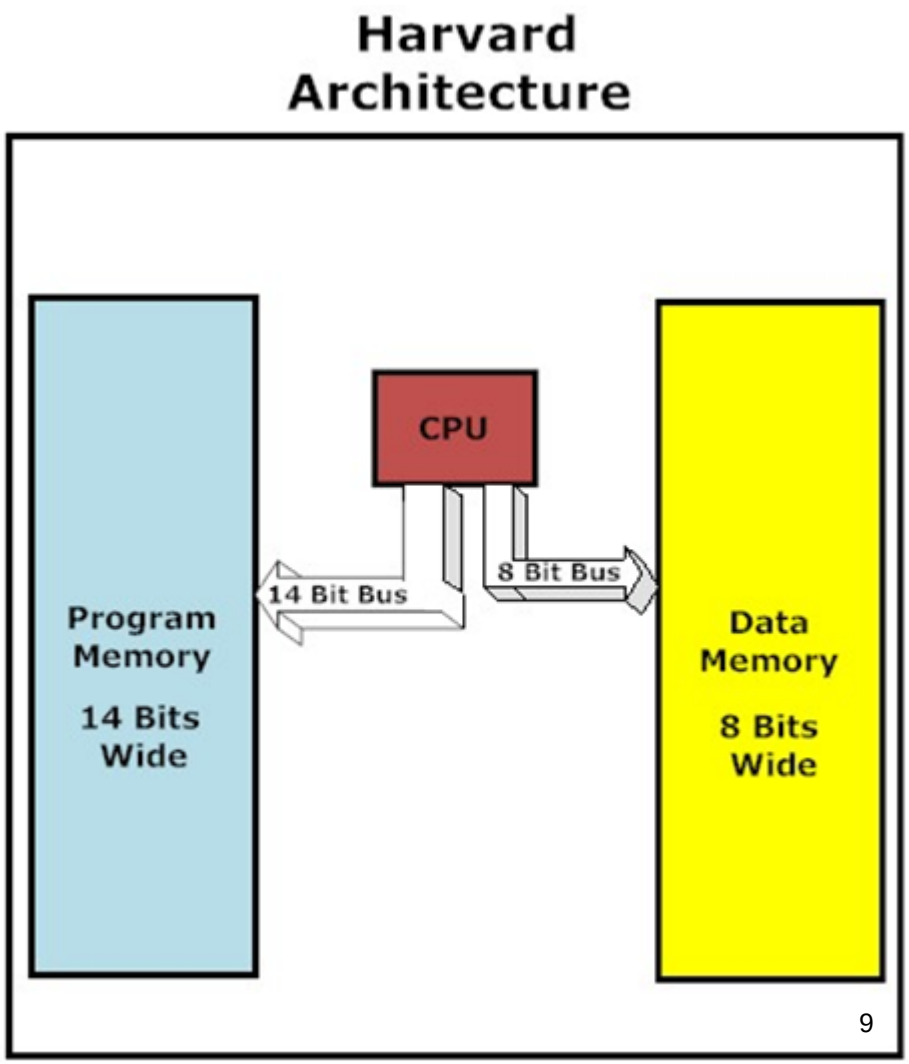
Some RISC V Pseudo Instructions

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero

Some more are available…

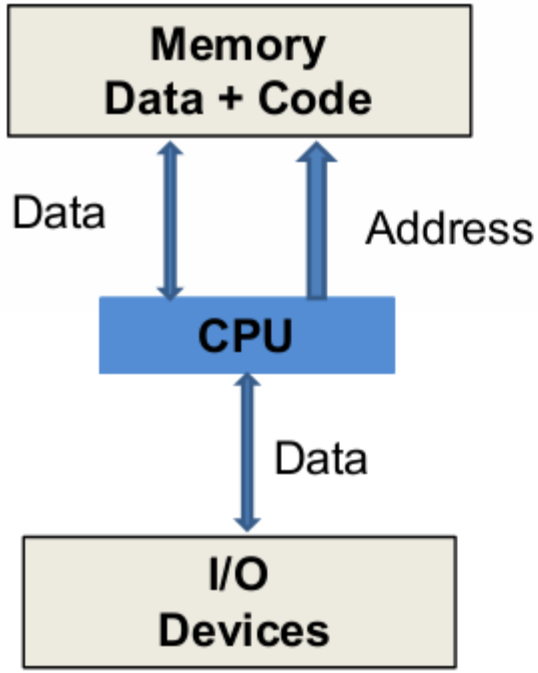
Memory Architectures



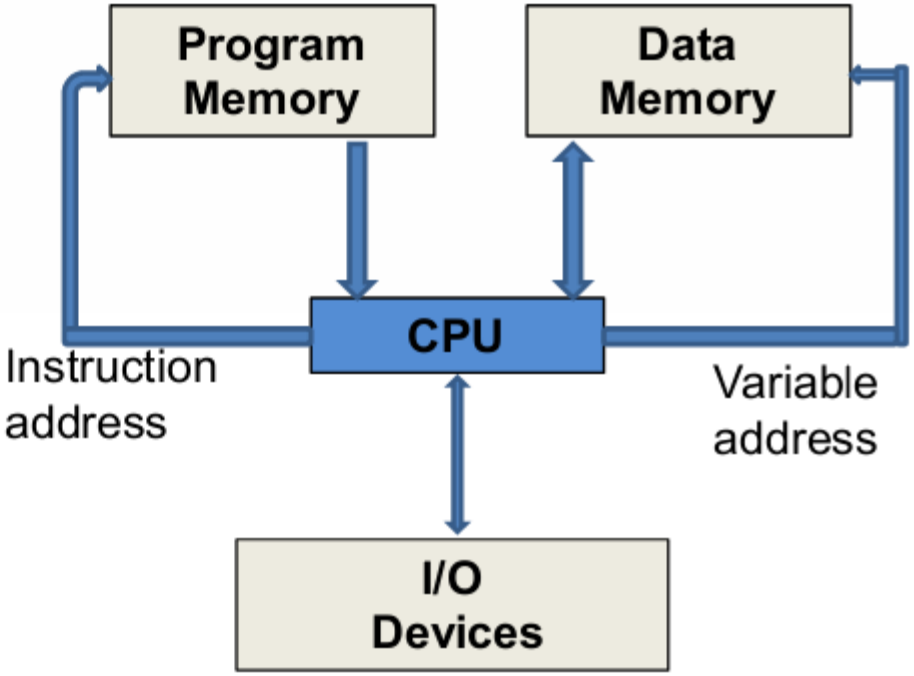


Source: Introduction to PIC Microcontroller – Part 1 by Khan Wahid

Von Neumann vs. Harvard Architecture

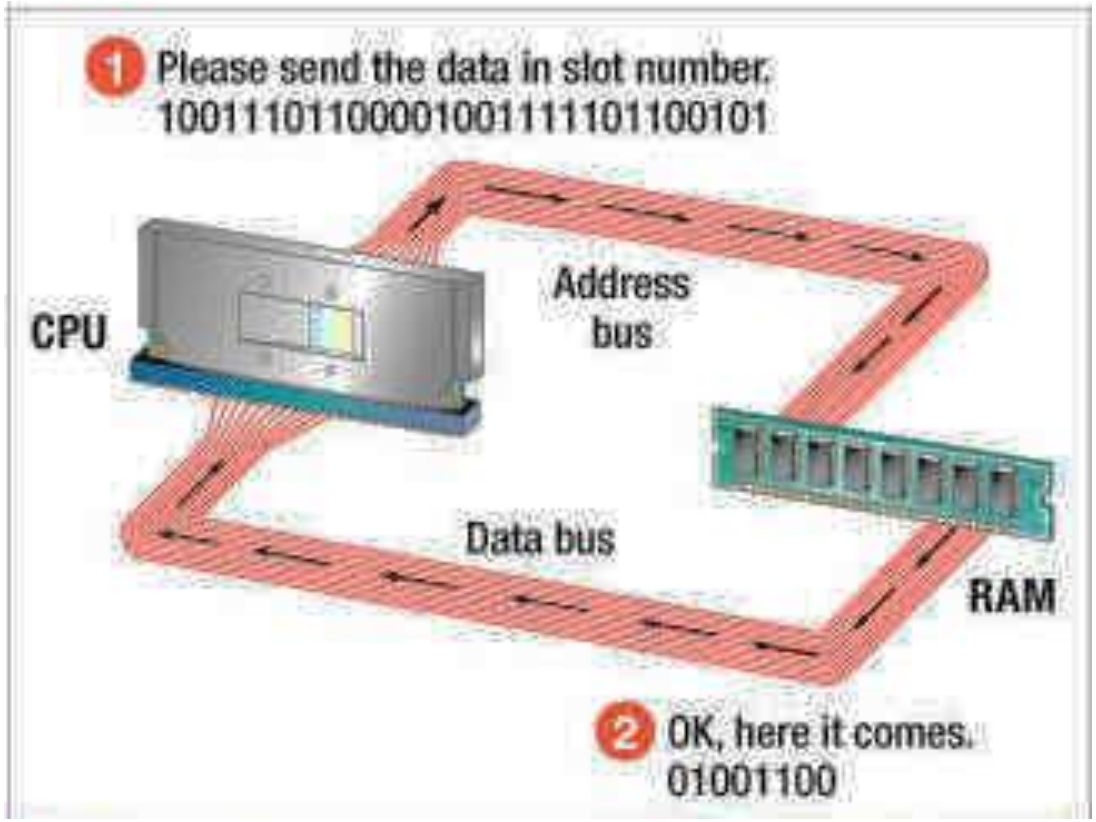


Von Neumann Machine



Harvard Machine

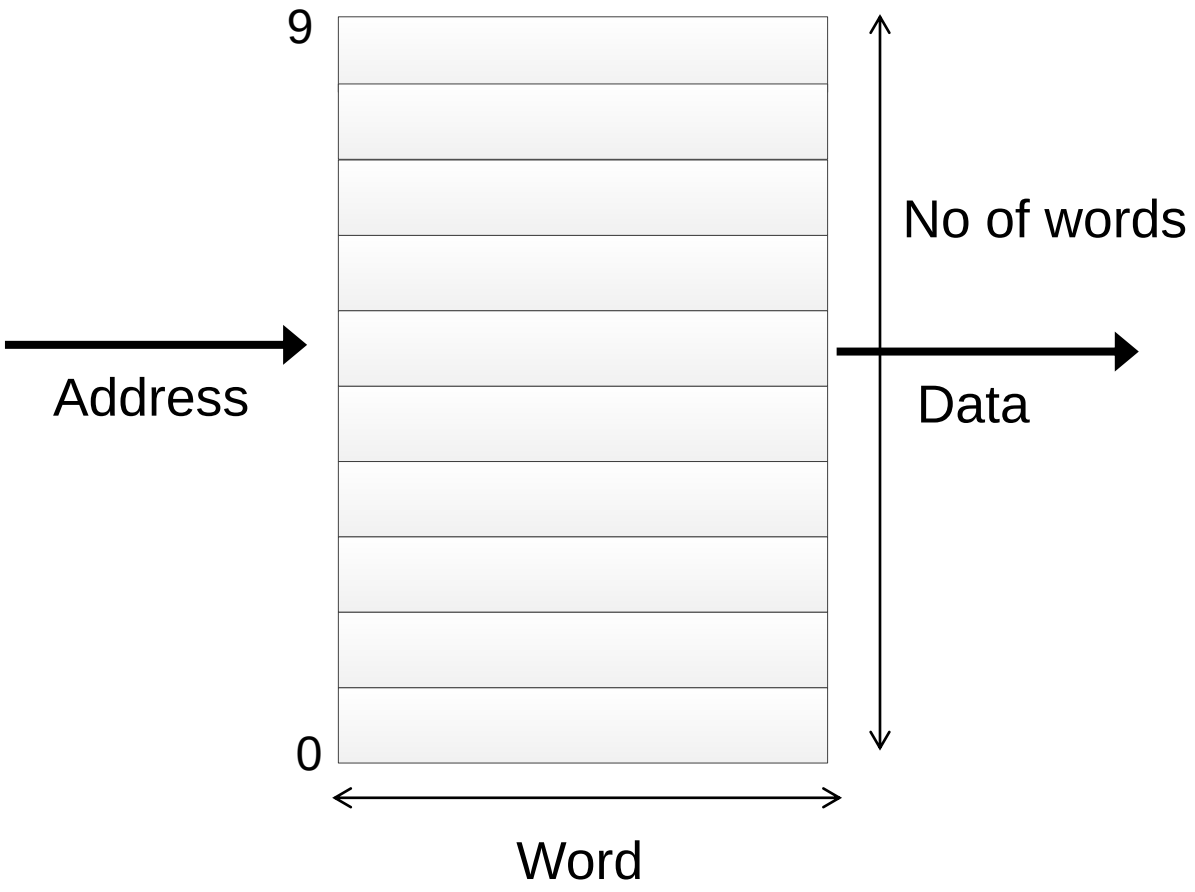
Memory Addressing



- Place an address on address bus
- Read or write operation
- Data placed on data bus

Source: www.alf.sd83.bc.ca/courses/It12/using_it/processor_speed.htm

Memory Addressing (Cont.)



Addressing Modes

- It is the way microprocessor:
  - Identifies location of data
  - Access data
- Absolute address
  - Actual physical address
  - Direct addressing
- Relative address
  - Address relative to a known reference
  - Indirect addressing

Load and Store Instructions

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Load: copy a value from memory to register *rd* : (Immediate type)
Source is a **memory address**
Store: copy the value in register *rs2* to memory : (Store type)

Need the proper 32bit address

RV32I Base Integer Instructions (Immediate type and Store type)

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]	

Activity

```
.data
    A: .word 0x1F2F3F4F
.text

.globl main
main:
    la a0, A
    la a0, A
    la a0, A
    ret
.end
```

- Try the following program with RIPES and explain the machine code
- What is the machine code of la a0, A ?
- Try to add a nop before first la and revisit machine code

la (Load Address) pseudo instruction

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address

```
.data
    A: .word 0x1F2F3F4F
.text

.globl main
main:
    la a0, A
    la a0, A
    la a0, A
    ret
.end
```

```
000000d8 <main>:
  d8:  00002517    auipc    a0,0x2
  dc:  0b050513    addi     a0,a0,176 # 2188 <A>
  e0:  00002517    auipc    a0,0x2
  e4:  0a850513    addi     a0,a0,168 # 2188 <A>
  e8:  00002517    auipc    a0,0x2
  ec:  0a050513    addi     a0,a0,160 # 2188 <A>
  f0:  00008067    ret
```

- We wrote the same instruction, but it is converted to different immediate values by assembler

Branching Instructions

Branching Instructions

RV32I Base Integer Instructions (Branch type)

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

- Change the Program Counter

  ⇔ Change the Program Flow

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Branching Instructions

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- Need to specify an address to go to
- Also take two registers to compare
  - Conditional branch
- Doesn't write into a register (similar to stores) (destination register *rd* is not required)
- How to encode label, i.e., where to branch to?

Branching Instructions -Addressing

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- PC-relative addressing
  - Use immediate field **12 bits** as a two's complement offset to PC.
  - Branches generally change the PC only by a small amount. But what is the reach?
  - Can specify $[-2^{11}, 2^{11})$ address offsets from the PC

Branching Instructions -Addressing

- Recall: RISCV uses 32-bit addresses, and memory is **byte-addressed**
- Instructions are ***"word-aligned"***: Address is always a multiple of 4 (in bytes)
  - Previous instruction offset : -4 bytes
  - Next instruction offset : 4 bytes
  - Only if compressed instructions (16 bytes) are used, it could be multiple of 2 (in bytes) as well
- PC ALWAYS points to an instruction
  - PC is typed as a pointer to a word
  - can do C-like pointer arithmetic

Branching Instructions -Addressing

- Only 12 bits available for immediate offset
- Offset Range $[-2^{11}, 2^{11})$

Offset : -4

1	1	1	1	1	1	1	1	1	1	1	0	0
12	11	10	9	8	7	6	5	4	3	2	1	0

Ignore the lsb, always zero (only 12 bits)

12 bit Immediate field (Two's complement)

Provision for compressed instructions (For normal instructions, this also will be always zero)

- Offset Range $[-2^{12}, 2^{12})$

Offset : -4

12 bit Immediate field

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

We don't do this

If we ignore both lsb's we will have more reach, but compressed instructions cannot be accommodated

Branch Calculation

- If we don't take the branch:
  - `PC = PC+4 =` next instruction
    - Assume we have only normal 32bit instructions
- If we do take the branch:
  - `PC = PC + (immediate field *2)`
    - If we assume we only have normal 32bit instructions, lsb of immediate field must be zero. If its one, there will be an error, by PC pointing to a middle of the instruction

Branching Instructions (B type)

```
main:
    addi t0,zero,10
    add t1,zero,zero
repeat:
    addi t1,t1,1
    bne t0,t1,repeat
    ret
.end
```

```
bne x5,x6,repeat
bne t0,t1,repeat
```

Offset : -4 = -000100
= 111011 +1
= 111100 (Sign extended)

1	1	1	1	1	1	1	1	1	1	1	0	0
12	11	10	9	8	7	6	5	4	3	2	1	0

31 20 19 15 14 12 11 7 6 0

Imm[12|10:5]	rs2	rs1	funct3	Imm[4:1|11]	opcode
Immediate offset [12|10:5]	rs2	rs1	funct3 sw	Immediate offset [4:1|11]	B type branch
			0 0 1		1 1 0 0 0 1 1
1 1 1 1 1 1 1	0 0 1 1 0	0 0 1 0 1		1 1 1 0 1	
1 1 1 1 1 1 1 0 0 1 1 0	0 0 1 0 1	0 0 1	1 1 1 0 1	1 1 0 0 0 1 1	
f	e	6	2	9	e	e	3

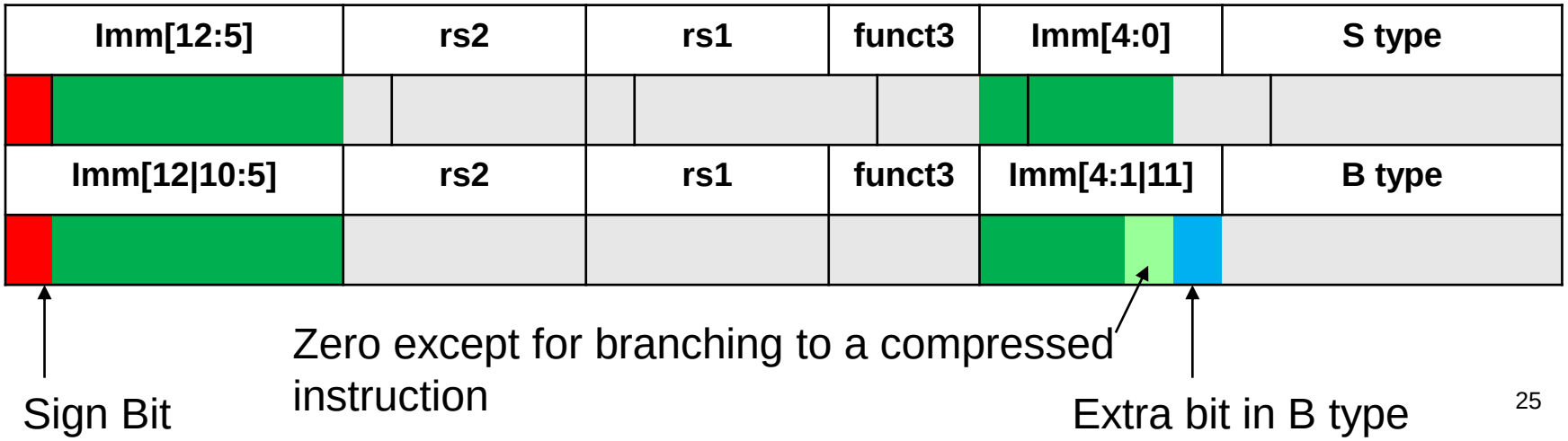
Machine Instruction: **0xFE629EE3**

https://stackoverflow.com/questions/58414772/why-are-risc-v-s-b-and-u-j-instruction-types-encoded-in-this-way

Branching Instructions -Addressing

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- S type and B type immediate value bits are aligning well.



Branching Instructions -Addressing

- Pseudo Assembly /Disassembly

```
main:
    addi t0,zero,10
    add t1,zero,zero
repeat:
    addi t1,t1,1
    bne t0,t1,repeat
    ret
.end
```

PC	Machine Code	Basic Code	Original Code
0x0	0x00A00293	addi x5 x0 10	addi t0,zero,0xa
0x4	0x00000333	add x6 x0 x0	add t1,zero,zero
0x8	0x00130313	addi x6 x6 1	addi t1,t1,0x1
0xc	0xFE629EE3	bne x5 x6 -4	bne t0,t1,repeat
0x10	0x00008067	jalr x0 x1 0	ret

```
000000d8 <main>:
  d8: 00a00293          li  t0,10
  dc: 00000333          add t1,zero,zero

000000e0 <repeat>:
  e0: 00130313          addi  t1,t1,1
  e4: fe629ee3          bne t0,t1,e0 <repeat>
  e8: 00008067          ret
```

Remember

`bne t0,t1,repeat`

Machine Instruction: **0xFE629EE3**

Jump Instructions

Jump Instructions

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Jump to anywhere in memory

Need the proper 32bit address

Store a return address in a register (rd), so that you can come back to original flow

Known as "Linking"

RV32I Base Integer Instructions (Jump type and Immediate type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
`jal`	Jump And Link	J	1101111			`rd = PC+4; PC += imm`	
`jalr`	Jump And Link Reg	I	1100111	0x0		`rd = PC+4; PC = rs1 + imm`	

Jump Instructions

- For branches, we assumed that we won't want to branch too far, so we can specify a change in the PC
- For (`jalr`) jumps, we may jump to anywhere in code memory
  - Ideally, we would specify a 32-bit memory address to jump to
  - Unfortunately, we can't fit both a 7-bit `opcode` and a 32-bit address into a single 32-bit word
  - Also, when linking we must write to an `rd` register

jal Instruction and j pseudo instruction

- `jal` saves PC+4 in register `rd` (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
- "j" jump is a pseudo-instruction—the assembler will instead use `jal` but sets rd=x0 to discard return address
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

jal Instruction Encoding

imm[20|10:1|11|19:12]	rd	opcode	J-type

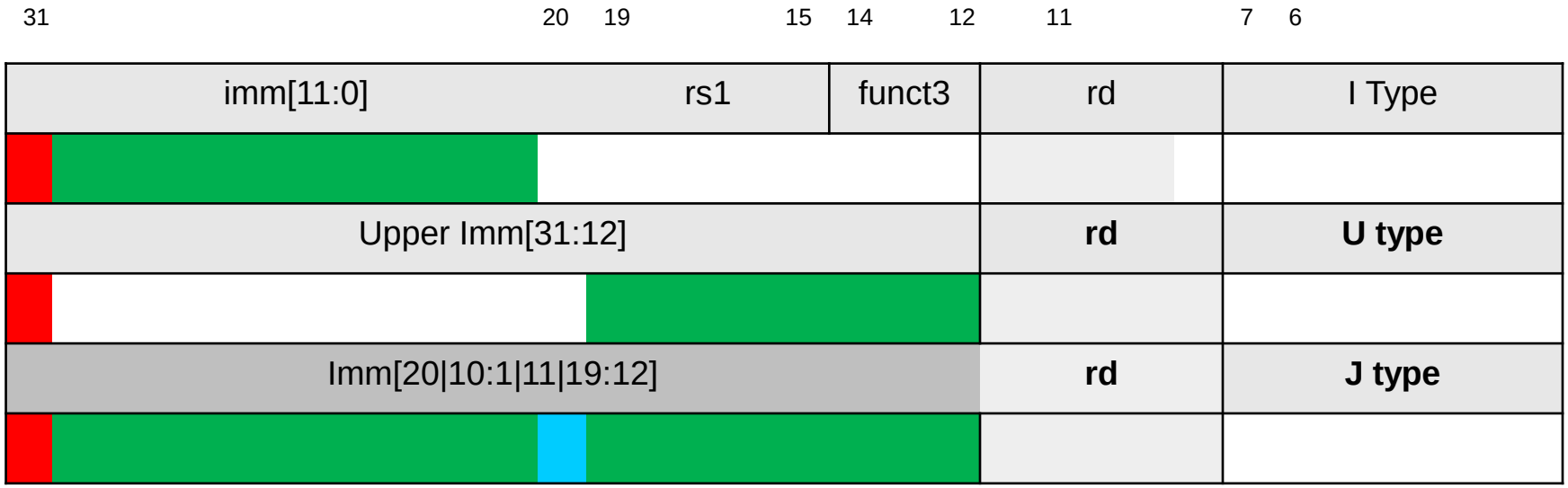
```
jal x1,-12
jal ra,myfunction
```

```
-12 = -001100
    =  110011 +1
= … 1111 1111 1111 0100
```

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
Imm [20|10:1|11|19:12]			**rd**	**opcode**
Immediate offset [20|10:1|11|19:12]			Register for return address	J type jump jal
				1 1 0 1 1 1 1
1 1 1 1 1 1 1 1 0 1 0 1	1 1 1 1 1	1 1 1	0 0 0 0 1	
			0 0 0 0 1	1 1 0 1 1 1 1
F	F	5	F	F	0	E	F

Machine Instruction: **0xFF5FF0EF**

Re-ordering bits in J type



jal Instruction Usage

```
.globl main
myfunc:
    addi t0, zero, 1
    ret

main:
    addi t0, zero, 0
    jal ra,myfunc
    ret
.end
```

PC	Machine Code	Basic Code	Original Code
0x0	0x00100293	addi x5 x0 1	addi t0, zero, 1
0x4	0x00008067	jalr x0 x1 0	ret
0x8	0x00000293	addi x5 x0 0	addi t0, zero, 0
0xc	0xFF5FF0EF	jal x1 -12	jal ra,myfunc
0x10	0x00008067	jalr x0 x1 0	ret

```
000000d8 <myfunc>:
  d8: 00100293          li  t0,1
  dc: 00008067          ret

000000e0 <main>:
  e0: 00000293          li  t0,0
  e4: ff5ff0ef          jal ra,d8 <myfunc>
  e8: 00008067          ret
```

jalr instruction (I type)

- Jump and link register `jalr` to address given in rs1 register + imm offset

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

RISCV Instructions:

```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0

# Call function at any 32-bit absolute address
lui x1, <hi 20 bits>
jalr ra, x1, <lo 12 bits>

# Jump PC-relative with 32-bit offset
auipc x1, <hi 20 bits>
jalr x0, x1, <lo 12 bits>
```

jalr Instruction encoding

```
jalr x1, x1, 0

jalr ra, ra, 0
```

imm[11:0]	rs1	funct3	rd	Opcode (jalr)
		0 0 0		1 1 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1		0 0 0 0 1	
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1	0 0 0	0 0 0 0 1	1 1 0 0 1 1 1

0	0	0	0	8	0	E	7

Machine Instruction: **0x000080E7**

Linking Multiple Functions

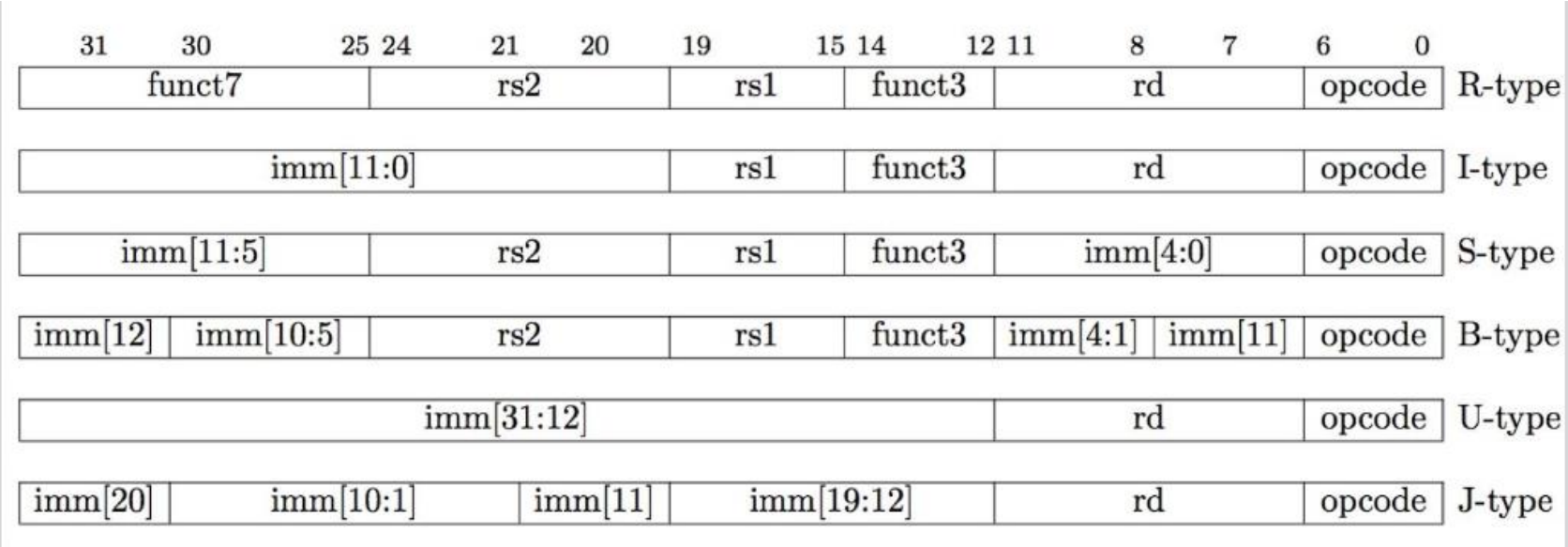
```
.globl main
myfunc:
    addi t0, zero, 1
    ret

main:
    addi t0, zero, 0
    la ra,myfunc
    jalr ra,ra,0
    ret
.end
```

```
.globl main
myfunc:
    addi t0, zero, 1
    ret

main:
    addi t0, zero, 0
    la ra,myfunc
    jalr ra,ra,0
    ret
.end
```

PC	Machine Code	Basic Code	Original Code
0x0	0x00100293	addi x5 x0 1	addi t0, zero, 1
0x4	0x00008067	jalr x0 x1 0	ret
0x8	0x00000293	addi x5 x0 0	addi t0, zero, 0
0xc	0x00000097	auipc x1 0	la ra,myfunc
0x10	0xFF408093	addi x1 x1 -12	la ra,myfunc
0x14	0x000080E7	jalr x1 x1 0	jalr ra,ra,0
0x18	0x00008067	jalr x0 x1 0	ret

```
000000d8 <myfunc>:
  d8: 00100293          li  t0,1
  dc: 00008067          ret

000000e0 <main>:
  e0: 00000293          li  t0,0
  e4: 00000097          auipc ra,0x0
  e8: ff408093          addi  ra,ra,-12 # d8 <myfunc>
  ec: 000080e7          jalr  ra
  f0: 00008067          ret
```

Summary



- The Stored Program concept is very powerful
  - Instructions can be treated and manipulated the same way as data in both hardware and software

Thank you.

Example – Logic Operations

- Write an assembly program to convert a given character from uppercase to lowercase & vice versa
- If we consider ASCII, this can be achieved by changing the 5th bit
  - A = 65 =0x41 = 0 1 0 0 0 0 0 1
  - a = 97 =0x61 = 0 1 1 0 0 0 0 1
- Get XOR with 00100000 = 32 = 0x20

Homework

- Write an assembly program to multiply 3 & 4
- Steps:
  - How many registers?
  - What registers to use?
  - What instructions to use?

Exercise

- Which mask or filter value would you use, and what operation would you perform to make the 2nd & 4th bits one (1) no matter what they were before?

- Which mask or filter value would you use, and what operation would you perform to flip the 2nd & 4th bits no matter what they were before?

Exercise: Convert to Assembly

```
int total = 0;
for (int i=10 ; i!=0; i--)
{
    total += i;
}
```

Exercise

- Write a program to calculate the total of all integers from 1 to 10
- High-level program

```
int total = 0;
for (int i=1 ; i<=10; i++)
{
   total += i;
 }
```

Exercise (Cont.)

- Steps
  - Are there any conditions/loops?
  - How many registers?
  - What registers to use?
  - What instructions to use?

Summary

- Instruction Set Architecture (ISA) is the layer between hardware & software
- Specific to a given chip unless standardized
  - Defines registers it contain
  - Micro-operations performed on data stored on those registers
- Typical Assembly instructions for
  - Register operations
  - Memory access (Load/Store)
  - Upper immediates and Address calculations
  - Branching
  - Jump and Link
- Assembler translates human readable Assembly code to machine code

Sign extended immediate values

```
addi t1, t1, -0xFFFFFFFF        # = 1       # ||
addi t1, t1, -0xFFFFFFFE        # = 2       # ||
addi t1, t1, -0xFFFFFFFD        # = 3       # ||
addi t1, t1, -0xFFFFF801        # =  2047   # ||
# addi t1, t1, -0xFFFFF800      # =  2048 (Out of range)
# addi t1, t1, -0xFFFFF7FF      # =  2049 (Out of range)
# addi t1, t1, -0x00000801      # = -2049 (Out of range)
addi t1, t1, -0x00000800        # = -2048   # ||
addi t1, t1, -0x000007FF        # = -2047   # ||
addi t1, t1, -0x00000001        # = -1      # ||
addi t1, t1, 0x00000000         # = 0       # ||
addi t1, t1, 0x00000001         # = 1       # ||
addi t1, t1, 0x00000007FF       # = 2047    # ||
# addi t1, t1, 0x00000800       # = 2048  (Out of range)
# addi t1, t1, 0x00000801       # = 2049  (Out of range)
# addi t1, t1, 0xFFFFF7FF       # = -2049  (Out of range)
addi t1, t1, 0xFFFFF800         # = -2048
addi t1, t1, 0xFFFFFFFE         # = -2
addi t1, t1, 0xFFFFFFFF         # = -1
```

Thank you!

(Compare with RISC-V Instructions)

TABLE 13-2: PIC16F87X INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

F→ file register
W→ working register
B→ bit
L→ literal (number)
Z→ conditional execution
d→ destination bit
d=0 store in W
d=1 store in f
use , w or ,f instead

Source: Makis Malliris & Sabir Ghauri, UWE