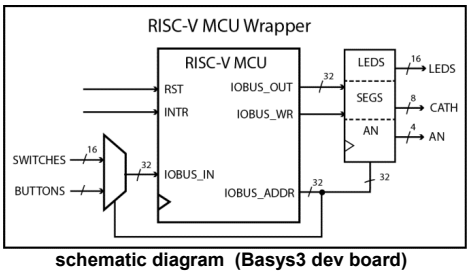
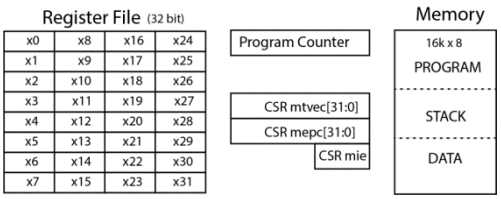
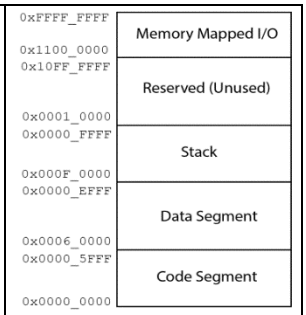
RISC-V MCU Architecture and Assembly Language Cheat Sheet



schematic diagram (Basys3 dev board)



programming model



RISC-V OTTER Memory Map

RISC-V OTTER Instruction Set: (shaded indicate pseudoinstructions)

Program Control					
jal	rd,imm	**j**	imm	**jal**	imm
jalr	rd,rs1,imm	**jr**	rs	**jalr**	rs
call	imm				
ret		**mret**			
beq	rs1,rs2,imm	**beqz**	rs1,imm		
bne	rs1,rs2,imm	**bnez**	rs1,imm		
blt	rs1,rs2,imm	**blez**	rs1,imm	**bgt**	rs1,rs2,imm
bge	rs1,rs2,imm	**bgez**	rs1,imm	**ble**	rs1,rs2,imm
bltu	rs1,rs2,imm	**bltz**	rs1,imm	**bgtu**	rs1,rs2,imm
bgeu	rs1,rs2,imm	**bgtz**	rs1,imm	**bleu**	rs1,rs2,imm

Load/Store (& I/O)					
lb	rd,imm(rs1)			**sb**	rs2,imm(rs1)
lh	rd,imm(rs1)			**sh**	rs2,imm(rs1)
lw	rd,imm(rs1)	**lw**	rd,imm	**sw**	rs2,imm(rs1)
lbu	rd,imm(rs1)				
lhb	rd,imm(rs1)				
Operations					
addi	rd,rs1,imm	**add**	rd,rs1,rs2		
		sub	rd,rs1,rs2	**neg**	rd,rs1
xori	rd,rs1,imm	**xor**	rd,rs1,rs2	**not**	rd,rs1
ori	rd,rs1,imm	**or**	rd,rs1,rs2		
andi	rd,rs1,imm	**add**	rd,rs1,rs2		
slli	rd,rs1,imm	**sll**	rd,rs1,rs2		
srli	rd,rs1,imm	**srl**	rd,rs1,rs2	**sgtz**	rd,rs1
srai	rd,rs1,imm	**sra**	rd,rs1,rs2	**sltz**	rd,rs1
slti	rd,rs1,imm	**slt**	rd,rs1,rs2	**snez**	rd,rs1
sltiu	rd,rs1,imm	**sltu**	rd,rs1,rs2	**seqz**	rd,rs1
Auxillary					
nop		**auipc**	rd,imm	**lui**	rd,imm
csrrw	rd,csr,rs1	**li**	rd,imm	**mv**	rd,rs
la	rd,imm	csrw	csr,rs1		

Register Designations:

reg	ABI	reg	ABI	reg	ABI	reg	ABI
x0	0	x8	so/fp	x16	a6	x24	s8
x1	ra	x9	s1	x16	a7	x25	s9
x2	sp	x10	a0	x18	s2	x26	s10
x3	gp	x11	a1	x19	s3	x27	s11
x4	tp	x12	a2	x20	s4	x28	t4
x5	t0	x13	a3	x21	s5	x29	t4
x6	t1	x14	a4	x22	s6	x30	t5
x7	t2	x15	a5	x23	s7	x31	t6

x0 read only; set to zero

Interrupt Architecture:

Firmware:

Init:
```
csrw    mtvec,x6  # store ISR address
csrw    mie,x1    # enable interrupts
```

Exit ISR `mret`

Hardware:

On interrupt: CSR[mie] ← 0 (mask interrupt)
CSR[mepc] ← PC (save PC)
PC ← CSR[mtvec] (load interrupt vector)

Exiting ISR: PC ← CSR[mepc] (load return address)

```
dir_1:   .word   0x23, 0x47 # also .half, .byte
dir_2:   .space  34     # room for this many bytes
.text     # instructions follow this
.data     # data follows this (.word, .half, .byte)
```

- Max program size: ~16k instructions (32-bit/instr)
- 32 32-bit general purpose registers (GPRs)
- Register x0: read only; always zero
- Stack: implemented in memory

I/O (memory mapped): addrs≥ to 0xC000_0000

- Input
```
li      x10,0xC0000004  # put port addr in reg
lw      x20,0(x10)      # input data to x20
```
- Output
```
li      x10,0xC0000004  # put port addr in reg
sw      x20,0(x20)      # output data in x20
```

bit setting: OR with '1'	bit clearing: AND with '0'	bit toggling: XOR with '1'
`ori x3,x3,0xF ;set 4 LSBs`	`andi x3,x3,0xF clear 28 MSBs`	`xori x3,x3,0x0F ;toggle 4 LSBs`

Useful Constructs:

PUSH (store word on stack)
```
addi    sp,sp,-4    # decrease stack pointer
sw      x8,0(sp)    # copy x8 to stack
```

POP (copy word from stack)
```
lw      x8,0(sp)    # copy value to x8
addi    sp,sp,4     # increase stack pointer
```

if/else construct

"If" x20=0, jump to zero;
"Else", do something, then jump over "if"

```
            beq     x20,x0,if       # check cond: branch when met  (if)
else:       add     x27,x28,x29     # -- "else" (dummy instruction)
            j       done            # jump over "if"
if:         sub     x17,x18,x19     # -- "if" (dummy instruction)
done:       nop                     # keep doing meaningful stuff
```

iterative construct
(unknown iterative #: `do-while`)

initialize x22 with check value
add to count each iteration
check result

```
init:       li      x22,923         # load some value to x20
            mv      x20,x0          # clear register
loop:       add     x20,x20,x23     # add some value
            blt     x20,x22,loop    # repeat if sum is < x22
loop_done:  nop                     # do something else…
```

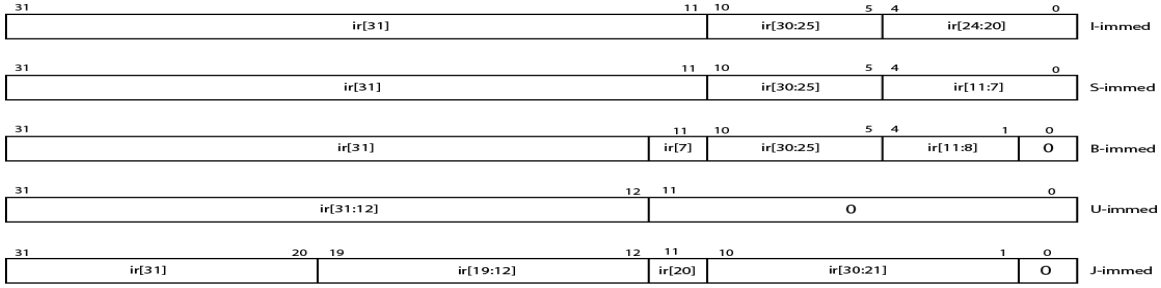
iterative construct
(known iterative value: `while`)

Initial loop count

```
init:       li      x20,10          # initialize loop count
loop:       beq     x20,x0,loop_done  # check stopping condition
            add     x4,x5,x6        # -- dummy instr (do something)
            addi    x20,x20,-1      # decrement iteration variable
            j       loop            #  branch to continue
loop_done:  nop                     # do something else…
```

Instruction	Description	RTL	Comment
add rd,rs1,rs2	addition	X[rd] ← X[rs1] + X[rs2]	
addi rd,rs1,imm	addition with immediate	X[rd] ← X[rs1] + sext(imm)	
and rd,rs1,rs2	bitwise AND	X[rd] ← X[rs1] · X[rs1]	
andi rd,rs1,imm	Bitwise AND immediate	X[rd] ← X[rs1] · sext(imm)	
auipc rd,imm	add upper immediate to PC	X[rd] ← PC + sext(imm)	
beq rs1,rs2,imm	branch if equal	PC ← PC + sext(imm) if (X[rs1] == X[rs2])	imm ≠ value
beqz rs1,imm	branch if equal to zero	PC ← PC + sext(imm) if (X[rs1] == 0)	imm ≠ value
bge rs1,rs2,imm	branch if greater than or equal	PC ← PC + sext(imm) if (X[rs1] $\geq_s$ X[rs2])	imm ≠ value
bgeu rs1,rs2,imm	branch if greater than or equal unsigned	PC ← PC + sext(imm) if (X[rs1] $\geq_u$ X[rs2])	imm ≠ value
bgez rs1,imm	branch if greater than or equal to zero	PC ← PC + sext(imm) if (X[rs1] $\geq_s$ 0)	imm ≠ value
bgt rs1,rs2,imm	branch if greater than	PC ← PC + sext(imm) if (X[rs1] $>_s$ X[rs2])	imm ≠ value
bgtu rs1,rs2,imm	branch if greater than unsigned	PC ← PC + sext(imm) if (X[rs1] $>_u$ X[rs2])	imm ≠ value
bgtz rs1,rs2,imm	branch if greater than zero	PC ← PC + sext(imm) if (X[rs1] $>_s$ 0)	imm ≠ value
ble rs1,rs2,imm	branch if less than or equal	PC ← PC + sext(imm) if (X[rs1] $\leq_s$ X[rs2])	imm ≠ value
bleu rs1,rs2,imm	branch if less than or equal (unsigned)	PC ← PC + sext(imm) if (X[rs1] $\leq_u$ X[rs2])	imm ≠ value
blez rs1,rs2,imm	branch if less than or equal zero	PC ← PC + sext(imm) if (X[rs1] $\leq_s$ 0)	imm ≠ value
blt rs1,rs2,imm	branch if less than	PC ← PC + sext(imm) if (X[rs1] $<_s$ X[rs2])	imm ≠ value
bltz rs1,imm	branch if less than zero	PC ← PC + sext(imm) if (X[rs1] $<_s$ 0)	imm ≠ value
bltu rs1,rs2,imm	branch if less than (unsigned)	PC ← PC + sext(imm) if (X[rs1] $<_u$ X[rs2])	imm ≠ value
bne rs1,rs2,imm	branch if not equal	PC ← PC + sext(imm) if (X[rs1] ≠ X[rs2])	imm ≠ value
bnez rs1,imm	branch if not equal to zero	PC ← PC + sext(imm) if (X[rs1] ≠ 0)	imm ≠ value
call label	branch to subroutine	X[rd] ← PC + 8; PC ← &symbol (rd=X1 if rd omitted)	imm ≠ value
csrrw rd,csr,rs1	control & status register read & write	X[rd] ← CSR[csr]; CSR[csr] ← rs1	
csrw csr,rs1	control & status register write	CSR[csr] ← rs1	
j imm	unconditional branch	PC ← PC + sext(imm)	imm ≠ value
jal rd,imm **jal imm**	unconditional branch with offset	X[rd] ← PC + 4; PC ← PC + sext(imm)	imm ≠ value rd=X1 if rd omitd
jalr rd,rs1,imm **jalr rs** **jalr rs,imm**	unconditional branch with offset & link	X[rd] ← PC+4; PC ← (X[rs1] + sext(imm)) & ~1	imm ≠ value rd=X1 if rd omitd
jr rs1	unconditional branch to register address	PC ← X[rs1]	
la rd,symbol	load absolute address of symbol	X[rd] ← &symbol	
lb rd,imm(rs1)	load byte	X[rd] ← sext(M[X[rs1] + sext(imm)] [7:0])	
lbu rd,imm(rs1)	load byte unsigned	X[rd] ← M[X[rs1] + sext(imm)] [7:0]	
lh rd,imm(rs1)	load halfword	X[rd] ← sext(M[X[rs1] + sext(imm)] [15:0])	
lhu rd,imm(rs1)	load halfword unsigned	X[rd] ← M[X[rs1] + sext(imm)] [15:0]	
li rd,imm	load immediate	X[rd] ← imm	
lw rd,imm(rs1)	load word into register	X[rd] ← M[X[rs1] + sext(imm)] [31:0]	
lui rd,imm	load upper immediate	X[rd] ← imm[31:12] << 12	
mret	machine mode exception return	PC ← CSR[mepc]	
mv rd,rs1	move	X[rd] ← X[rs1]	
neg rd,rs2	negate	X[rd] ← -X[rs2]	
nop	no operation	nada (PC ← PC + 4)	
not rd,rs2	ones complement	X[rd] ← ~X[rs2]	
or rd,rs1,rs2	bitwise inclusive OR	X[rd] ← X[rs1] | X[rs2]	
ori rd,rs1,imm	bitwise inclusive OR immediate	X[rd] ← X[rs1] | sext(imm)	
ret	return from subroutine	PC ← X1	
sb rs2,imm(rs1)	store byte in memory	M[X[rs1] + sext(imm)] ← X[rs2][7:0]	
seqz rd,rs1	set if equal to zero	X[rd] ← (X[rs1] == 0) ? 1 : 0	
sgtz rd,rs2	set if greater than zero	X[rd] ← (X[rs2] $>_s$ 0) ? 1 : 0	
sh rs2,imm(rs1)	store halfword in memory	M[X[rs1] + sext(imm)] ← X[rs2][15:0]	
sw rs2,imm(rs1)	store word	M[X[rs1] + sext(imm)] ← X[rs2]	
sll rd,rs1,rs2	logical shift left	X[rd] ← X[rs1] << X[rs2]	
slli rd,rs1,shft_amt	logical shift left immediate	X[rd] ← X[rs1] << shft_amt	
slt rd,rs1,rs2	set if less than	X[rd] ← (X[rs1] $<_s$ X[rs2]) ? 1 : 0	
slti rd,rs1,imm	set if less than immediate	X[rd] ← (X[rs1] $<_s$ sext(imm)) ? 1 : 0	
sltiu rd,rs1,imm	set if less than immediate unsigned	X[rd] ← (X[rs1] $<_u$ sext(imm)) ? 1 : 0	
sltu rd,rs1,rs2	set if less than unsigned	X[rd] ← (X[rs1] $<_u$ X[rs2]) ? 1 : 0	
sltz rd,rs1	set if less than zero	X[rd] ← (X[rs1] $<_s$ 0) ? 1 : 0	
snez rd,rs2	set if not equal to zero	X[rd] ← (X[rs2] ≠ 0) ? 1 : 0	
sra rd,rs1,rs2	arithmetic shift right	X[rd] ← X[rs1] $>>_s$ X[rs2]	
srai rd,rs1,shft_amt	arithmetic shift right immediate	X[rd] ← X[rs1] $>>_s$ shft_amt	
srl rd,rs1,rs2	logical shift right	X[rd] ← X[rs1] >> X[rs2]	
srli rd,rs1,imm	logical shift right immediate	X[rd] ← X[rs1] >> imm	
sub rd,rs1,rs2	subtract	X[rd] ← X[rs1] – X[rs2]	
xor rd,rs1,rs2	exclusive OR	X[rd] ← X[rs1] ^ X[rs2]	
xori rd,rs1,imm	exclusive OR immediate	X[rd] ← X[rs1] ^ sext(imm)	

(shading indicates pseudoinstructions)

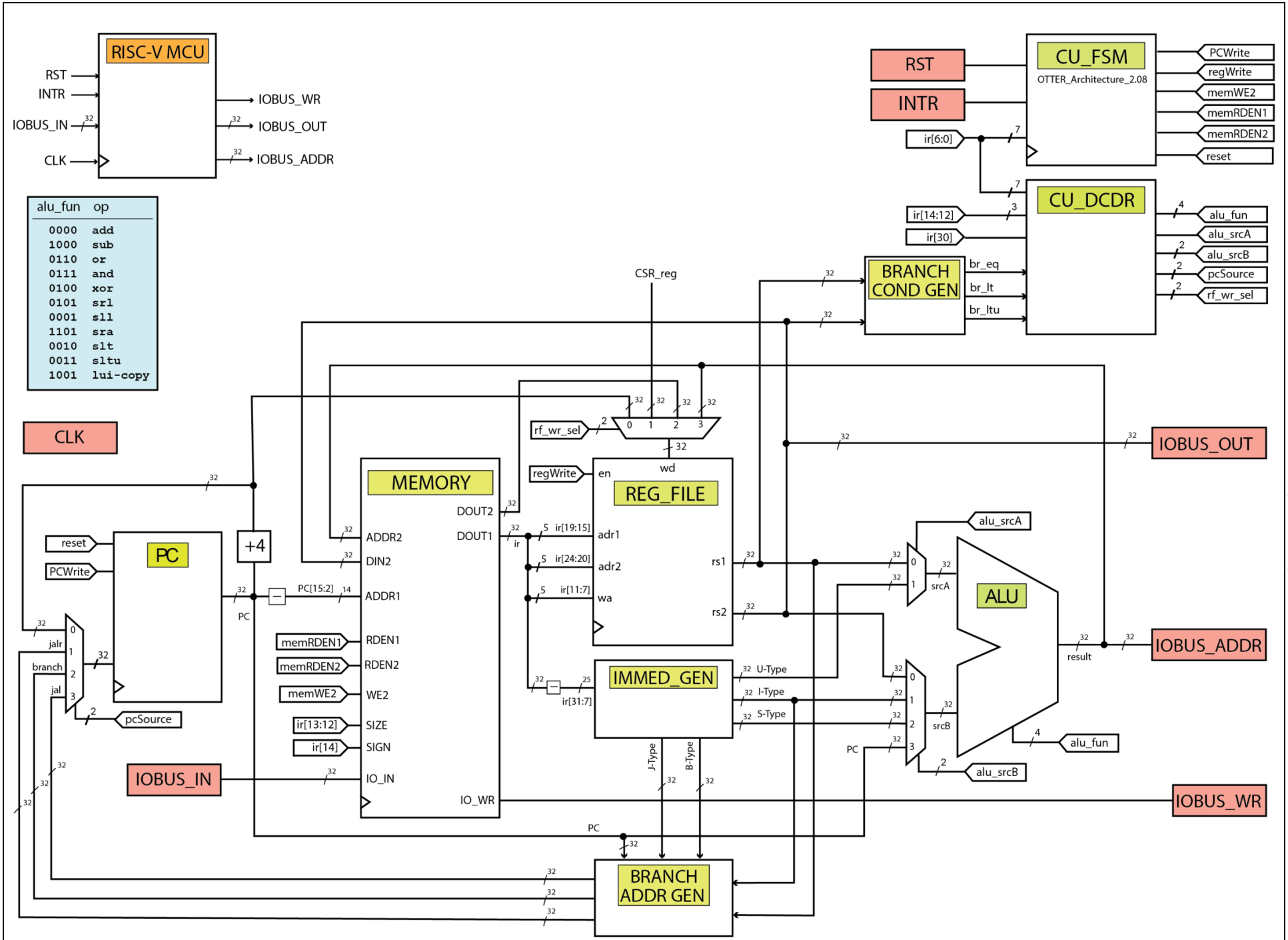


Instruction Formats

31 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

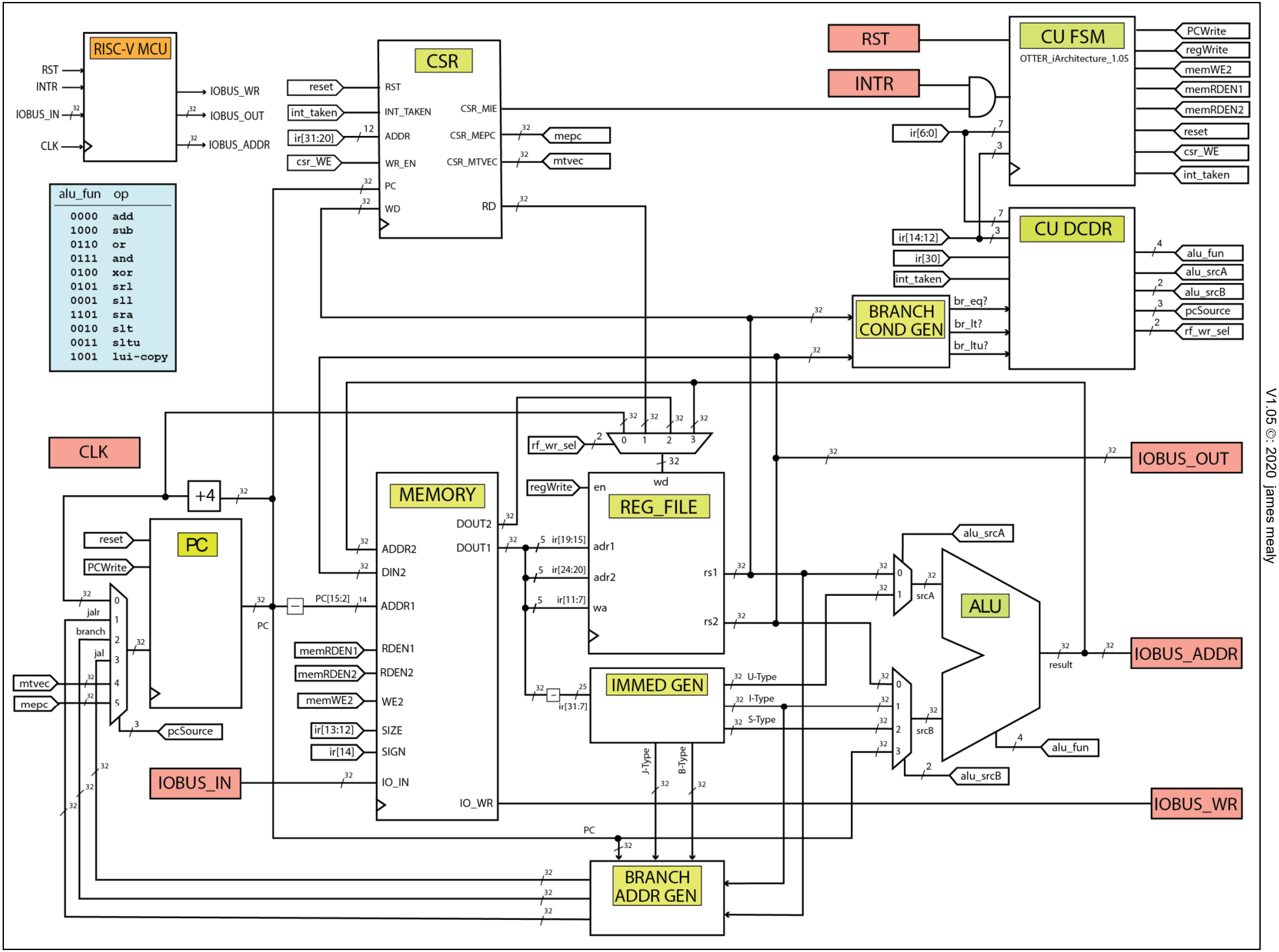
RISC-V Base Instruction Set

imm[31:12]			rd	0110111	LUI	U
imm[31:12]			rd	0010111	AUIPC	U
imm[20,10:1,11,19:12]			rd	1101111	JAL	J
imm[11:0]		rs1	000	rd	1100111	JALR	I
imm[11:0]		rs1	000	rd	0000011	LB	I
imm[11:0]		rs1	001	rd	0000011	LH	I
imm[11:0]		rs1	010	rd	0000011	LW	I
imm[11:0]		rs1	100	rd	0000011	LBU	I
imm[11:0]		rs1	101	rd	0000011	LHU	I
imm[11:0]		rs1	000	rd	0010011	ADDI	I
imm[11:0]		rs1	010	rd	0010011	SLTI	I
imm[11:0]		rs1	011	rd	0010011	SLTIU	I
imm[11:0]		rs1	110	rd	0010011	ORI	I
imm[11:0]		rs1	100	rd	0010011	XORI	I
imm[11:0]		rs1	111	rd	0010011	ANDI	I
imm[11:5]	*imm[4:0]	rs1	001	rd	0010011	SLLI	I
0000000	*imm[4:0]	rs1	101	rd	0010011	SRLI	I
0100000	*imm[4:0]	rs1	101	rd	0010011	SRAI	I
imm[12,10:5]	rs2	rs1	000	imm[4:1,11]	1100011	BEQ	B
imm[12,10:5]	rs2	rs1	001	imm[4:1,11]	1100011	BNE	B
imm[12,10:5]	rs2	rs1	100	imm[4:1,11]	1100011	BLT	B
imm[12,10:5]	rs2	rs1	101	imm[4:1,11]	1100011	BGE	B
imm[12,10:5]	rs2	rs1	110	imm[4:1,11]	1100011	BLTU	B
imm[12,10:5]	rs2	rs1	111	imm[4:1,11]	1100011	BGEU	B
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB	S
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH	S
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW	S
0000000	rs2	rs1	000	rd	0110011	ADD	R
0100000	rs2	rs1	000	rd	0110011	SUB	R
0000000	rs2	rs1	001	rd	0110011	SLL	R
0000000	rs2	rs1	010	rd	0110011	SLT	R
0000000	rs2	rs1	011	rd	0110011	SLTU	R
0000000	rs2	rs1	100	rd	0110011	XOR	R
0000000	rs2	rs1	101	rd	0110011	SRL	R
0100000	rs2	rs1	101	rd	0110011	SRA	R
0000000	rs2	rs1	110	rd	0110011	OR	R
0000000	rs2	rs1	111	rd	0110011	AND	R
csr		rs1	001	rd	1110011	CSRRW	sys
0011000	01000	0000	000	00000	1110011	MRET	sys



RISC-V OTTER MCU Architecture Diagram

V2.08 ©: 2020 james mealy



RISC-V OTTER MCU Architecture Diagram with Interrupts