RISC-V Reference Sheet

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

31 25|24	20|19	15|14	12|11	7|6	0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-Type
imm[19:0]				rd	opcode	U-Type

R-Type Computational Instructions:

						Name	Mnemonic	Implementation
0000000	rs2	rs1	000	rd	0110011	ADD	ADD rd, rs1, rs2	`R[rd] = R[rs1] + R[rs2]`
0100000	rs2	rs1	000	rd	0110011	SUB	SUB rd, rs1, rs2	`R[rd] = R[rs1] - R[rs2]`
0000000	rs2	rs1	001	rd	0110011	Shift Left	SLL rd, rs1, rs2	`R[rd] = R[rs1] << R[rs2]`
0000000	rs2	rs1	010	rd	0110011	Set Less Than	SLT rd, rs1, rs2	`R[rd] = (R[rs1] < R[rs2]) ? 1:0`
0000000	rs2	rs1	100	rd	0110011	XOR	XOR rd, rs1, rs2	`R[rd] = R[rs1] ⊕ R[rs2]`
0000000	rs2	rs1	101	rd	0110011	Shift Right Logical	SRL rd, rs1, rs2	`R[rd] = R[rs1] >>> R[rs2]`
0100000	rs2	rs1	101	rd	0110011	Shift Right Arithmetic	SRA rd, rs1, rs2	`R[rd] = R[rs1] >>> R[rs2]`
0000000	rs2	rs1	110	rd	0110011	OR	OR rd, rs1, rs2	`R[rd] = R[rs1] | R[rs2]`
0000000	rs2	rs1	111	rd	0110011	AND	AND rd, rs1, rs2	`R[rd] = R[rs1] & R[rs2]`

I-Type Computational Instructions:

					Name	Mnemonic	Implementation
imm[11:0]	rs1	000	rd	0010011	Add Immediate	ADDI rd, rs1, imm	`R[rd] = R[rs1] + imm`
imm[11:0]	rs1	001	rd	0010011	Shift Left Immediate	SLLI rd, rs1, imm	`R[rd] = R[rs1] << imm`
imm[11:0]	rs1	010	rd	0010011	Set Less Than Immediate	SLTI rd, rs1, imm	`R[rd] = (R[rs1] < imm) ? 1:0`
imm[11:0]	rs1	100	rd	0010011	XOR Immediate	XORI rd, rs1, imm	`R[rd] = R[rs1] ⊕ imm`
imm[11:0]	rs1	101	rd	0010011	Shift Right Immediate	SRLI rd, rs1, imm	`R[rd] = R[rs1] >>> imm`
imm[11:0]	rs1	101	rd	0010011	Shift Right Arith Immediate	SRAI rd, rs1, imm	`R[rd] = R[rs1] >>> imm`
imm[11:0]	rs1	110	rd	0010011	OR Immediate	ORI rd, rs1, imm	`R[rd] = R[rs1] | sign_extend(imm)`
imm[11:0]	rs1	111	rd	0010011	AND Immediate	ANDI rd, rs1, imm	`R[rd] = R[rs1] & sign_extend(imm)`

Load and Store Instructions:

						Name	Mnemonic	Implementation
imm[11:0]		rs1	010	rd	0000011	Load Word	LW rd, rs1, imm	`R[rd] = Mem[imm+R[rs1]]`
imm[11:0]		rs1	001	rd	0000011	Load Double Word	LD rd, rs1, imm	`R[rd] = Mem[imm+R[rs1]]`
imm[11:0]		rs1	000	rd	0000011	Load Byte	LB rd, rs1, imm	`R[rd] = Mem[imm+R[rs1]]`
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	Store Word	SW rs2, rs1, imm	`Mem[imm+R[rs1]] = R[rs2]`
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	Store Double Word	SD rs2, rs1, imm	`Mem[imm+R[rs1]] = R[rs2]`
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	Store Byte	SB rs2, rs1, imm	`Mem[imm+R[rs1]] = R[rs2]`

U-Type Computational Instructions:

				Name	Mnemonic	Implementation
imm[19:0]	rd	0110111	Load Upper Immediate	LUI rd, imm	`R[rd] = imm << 12`

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

31 25|24 20|19 15|14 12|11 7|6 0						
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-Type
imm[19:0]				rd	opcode	U-Type

Load and Store Conditional:

00010	X	X	00000	rs1	010	rd	0101111
00011	X	X	rs2	rs1	010	rd	0101111

Note: X bits are not taught as part of CS 3410.

Name	Mnemonic	Implementation
Load Reserved (Word) Load Reserved (Double Word)	LR.W rd, (rs1) LR.D rd, (rs1)	`R[rd] = Mem[R[rs1]], reservation on Mem[R[rs1]]`
Store Conditional (Word) Store Conditional (Double Word)	SC.W rd, rs2, (rs1) SC.D rd, rs2, (rs1)	`if reserved M[R[rs1]]=R[rs2], R[rd]=0, else: R[rd] = 1`

Jump and Branch Instructions:

imm[19:0]				rd	1101111
imm[11:0]		rs1	000	rd	1100111
imm[11:5]	rs2	rs1	000	imm[4:0]	1100011
imm[11:5]	rs2	rs1	101	imm[4:0]	1100011
imm[11:5]	rs2	rs1	100	imm[4:0]	1100011
imm[11:5]	rs2	rs1	101	imm[4:0]	1100011
imm[11:5]	rs2	rs1	100	imm[4:0]	1100011
imm[11:5]	rs2	rs1	001	imm[4:0]	1100011

Name	Mnemonic	Implementation
Jump and Link	JAL rd, imm	`R[rd] = PC+4; PC = PC + imm << 1`
Jump and Link Register	JALR rd, rs1, imm	`R[rd] = PC+4; PC = (R[rs1] + imm)`
Branch Equal	BEQ rs1, rs2, imm	`PC = if (R[rs1] == R[rs2]) PC + imm << 1`
Branch Greater than or Equal	BGE rs1, rs2, imm	`PC = if (R[rs1] >=s R[rs2]) PC + imm << 1`
Branch Less Than	BLT rs1, rs2, imm	`PC = if (R[rs1] <s R[rs2]) PC + imm << 1`
Branch GT or Equal Unsigned	BGEU rs1, rs2, imm	`PC = if (R[rs1] >=u R[rs2]) PC + imm << 1`
Branch Less Than Unsigned	BLTU rs1, rs2, imm	`PC = if (R[rs1] <u R[rs2]) PC + imm << 1`
Branch Not Equal	BNE rs1, rs2, imm	`PC = if (R[rs1] != R[rs2]) PC + imm << 1`

RISC-V Register Conventions

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global data pointer	--
x4	tp	thread pointer	--
x5-x7	t0-t2	temporaries	Caller
x8	s0/fp	frame pointer	Callee
x9	s1	saved register	Callee
x10-x11	a0-a1	fn arguments / return values	Caller
x12-x17	a2-a7	function arguments	Caller
x18-x27	s2-s11	saved registers	Callee
x28-x31	t3-t6	temporaries	Caller

Name	Mnemonic	Implementation
Jump	J imm	`JAL x0, imm`
Jump Register	JR rs1	`JALR x0, rs, 0`
Branch Greater Than	BGT rs1, rs2, imm	`BLT rs2, rs1, imm`
Branch Less Than or Equal	BLE rs1, rs2, imm	`BGE rs2, rs2, imm`
Branch if Equal to Zero	BEQZ rs1, imm	`BEQ rs1, x0, imm`
Branch if Not Equal to Zero	BNEZ rs1, imm	`BNE rs1, x0 imm`
Negate	NEG rd, rs1	`SUB rd, x0, rs1`
NOP	NOP	`ADDI x0, x0, x0`
Return	RET	`jalr x0, x1, 0; auipc x6, offset[31:12]`
NOT	NOT rd, rs1	`ADDI rd, rs1, 0`
Load Immediate	LI rd, imm	Myriad Sequences
Move	MV rd, rs1	`ADDI rd, rs1, 0`
Set if Equal to Zero	SEQZ rd, rs1	`SLITU rd, rs1, 1`
Set if Equal not to Zero	SNEZ rd, rs1	`SLTU rd, x0, rs`

POSIX Threads Cheatsheet

All of the functions below return `0` on success and a non-zero error code on error.

Thread Management

Function	Description

`int pthread_create(`
`  pthread_t* thread,`
`  pthread_attr_t* attr,`
`  void *(*start_routine)(void*),`
`  void* arg`
`);`
	Creates a new thread, with attributes specified by `attr`, that executes the function `start_routine` provided with the argument `arg`. The thread's metadata is stored in `thread`.

`int pthread_join(`
`  pthread_t thread,`
`  void **value_ptr`
`);`
	Blocks until `thread` terminates. A pointer to the return value of the thread's routine is stored in `value_ptr`.

Mutual Exclusion Locks

Function	Description

`int pthread_mutex_init(`
`  pthread_mutex_t* mutex,`
`  pthread_mutexattr_t* attr`
`);`
	Initializes the mutex referenced by `mutex` with attributes specified by `attr`.

`int pthread_mutex_destroy(`
`  pthread_mutex_t* mutex`
`);`
	Destroys the mutex referenced by `mutex`. The mutex may be reinitialized later.

`int pthread_mutex_lock(`
`  pthread_mutex_t* mutex`
`);`
	Locks the mutex referenced by `mutex`. If the mutex is already locked, the calling thread shall block until the mutex becomes available.

`int pthread_mutex_unlock(`
`  pthread_mutex_t* mutex`
`);`
	Releases the mutex referenced by `mutex`.

Condition Variables

Function	Description

`int pthread_cond_init(`
`  pthread_cond_t* cond,`
`  pthread_condattr_t* attr`
`);`
	Initializes the condition variable referenced by `cond` with attributes specified by `attr`.

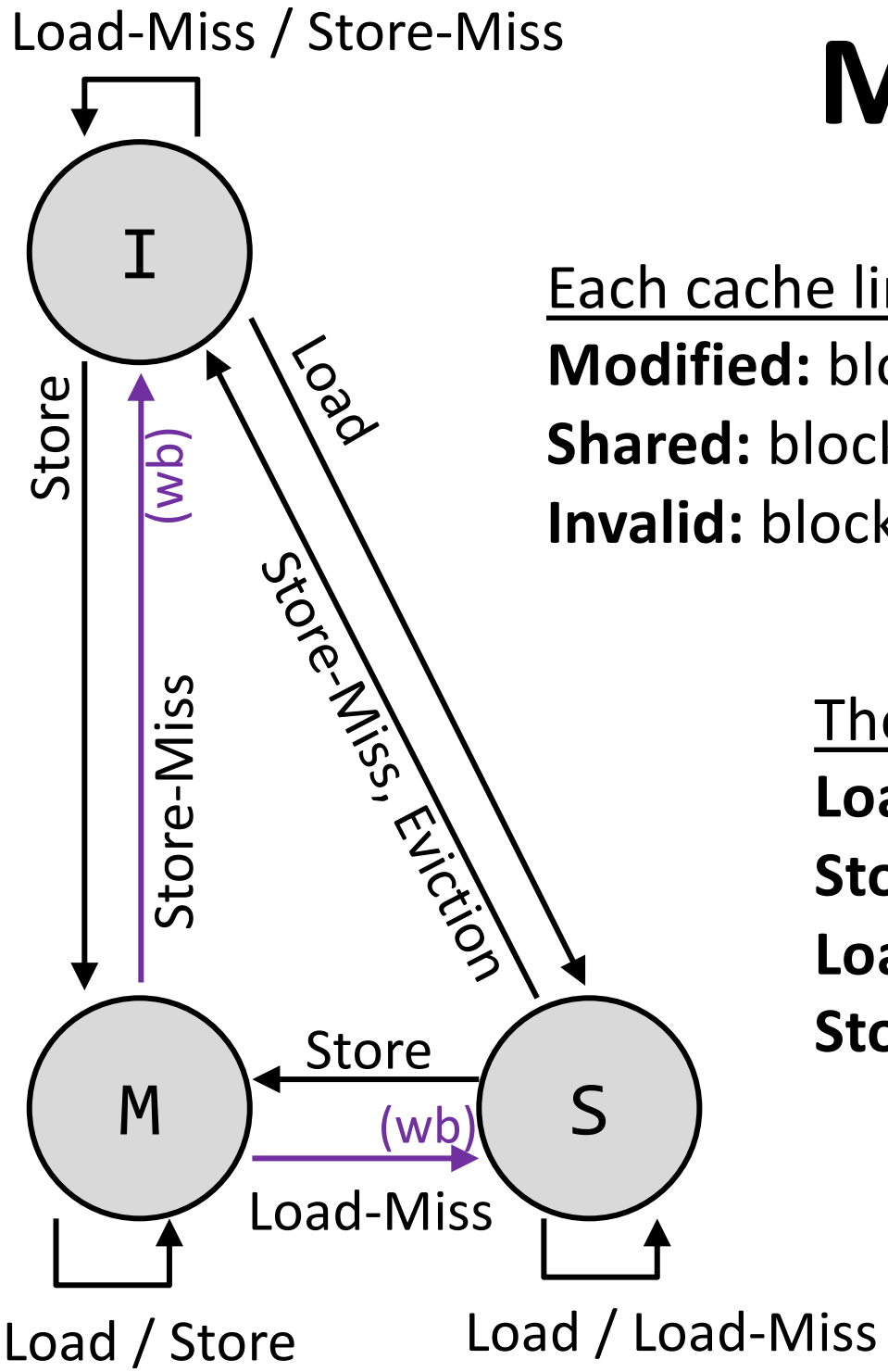
`int pthread_cond_destroy(`
`  pthread_cond_t* cond`
`);`
	Destroys the condition variable referenced by `cond`.

`int pthread_cond_wait(`
`  pthread_cond_t* cond,`
`  pthread_mutex_t* mutex`
`);`
	Atomically unlocks the `mutex` and waits for the condition variable `cond` to be signaled. Execution of the calling thread is suspended until the condition variable is signaled. The `mutex` must be locked by the calling thread before calling this function. Upon returning, the function re-acquires the `mutex`.

`int pthread_cond_signal(`
`  pthread_cond_t* cond`
`);`
	Restarts *one* of the threads that are waiting on the condition variable `cond`.

`int pthread_cond_broadcast(`
`  pthread_cond_t* cond`
`);`
	Restarts *all* of the threads that are waiting on the condition variable `cond`.

MSI protocol



Each cache line can be in 1 of 3 states:

Modified: block is in the cache, has been written

Shared: block is in the cache, has only been read

Invalid: block is not in the cache

There are 4 kinds of events:

Load by **the local** CPU

Store by **the local** CPU

Load-Miss by a remote CPU on the Bus

Store-Miss by a remote CPU on the Bus