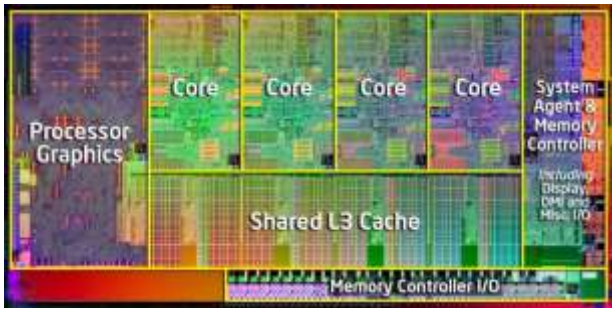
고려대학교
KOREA UNIVERSITY



COSE222 Computer Architecture

Lecture 4. RISC-V A Extension

Atomic Instructions

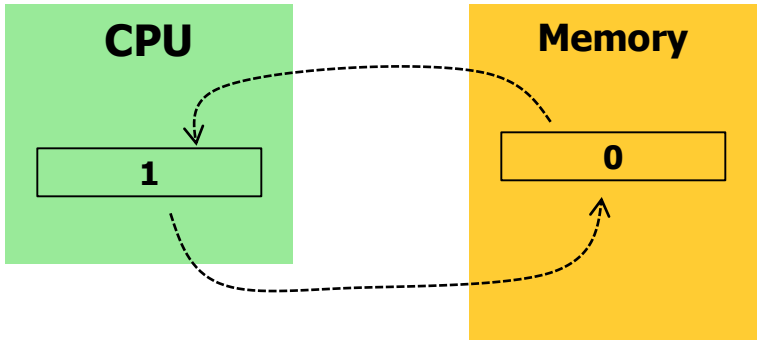
Prof. Taeweon Suh

Computer Science & Engineering

Korea University

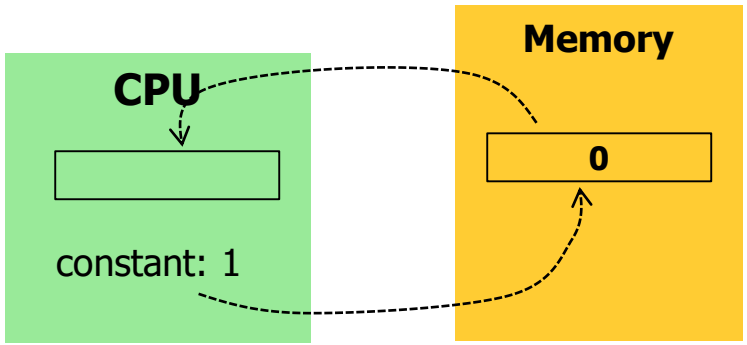
Atomic Read-Modify-Write Operation

- A typical approach: Atomic exchange instruction



Atomic Read-Modify-Write Operation

- Variants:
  - **test&set**
    - Read from memory to register, Set 1 to memory
    - Test register (0: success (free), 1: fail)




```
lock:    t&s   register, location
         bnz   register, lock
         ret

unlock:  st    location, #0
         ret
```

  - **swap**
    - Read from memory to register, store register to memory
    - Lock can be implemented by replacing **test&set** with **swap** as long as 0 and 1 are used

Atomic Read-Modify-Write Operation

- Variants:
  - **fetch&op** family
    - Read from memory location to register
    - Apply operation to the value read
    - Store the value to memory location
    - Examples: fetch&increment, fetch&decrement, fetch&add
  - **compare&swap (CAS)**
    - Compare the value in the mem location with first register
    - If equal, swap the value in memory and second register
    - Require 2 registers, a memory location

Lock-Free Data Structure with CAS

- For accessing shared resources, the traditional approach to multithreaded programming is to use locks
  - Synchronization primitives such as mutexes and semaphores are used to ensure that certain sections of code are not executed concurrently
  - Blocking a thread is undesirable for many reasons: deadlock, livelock, priority inversion

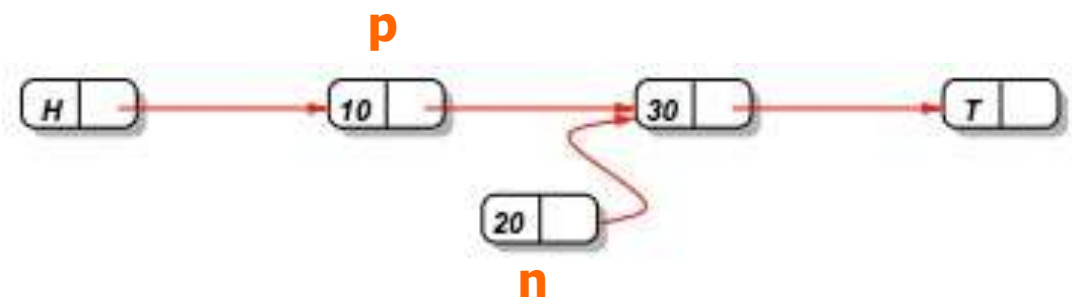
- Non-blocking algorithms do not suffer from these downsides
  - Lock-free data structure can be used to improve performance
  - Lock-free data structure increases the amount of time spent in parallel execution rather than serial execution
    - Access to the shared data structure does not need to be serialized to stay coherent

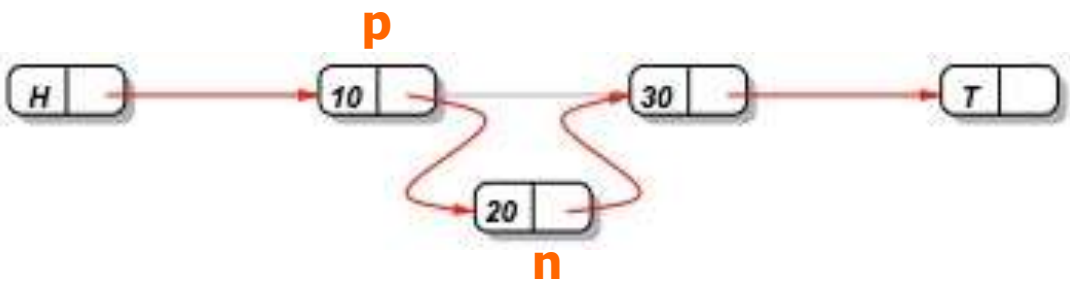
https://en.wikipedia.org/wiki/Non-blocking_algorithm

Example: Lock-free data structure w/ CAS

```
while (1) {
        n->next = p->next;
        Node *old_next = p->next;
        if (compare_and_swap(&p->next, old_next, n))
            return;
}
```

```
static int compare_and_swap(int *ptr, int oldVal, int newVal)
```





Atomic Read-Modify-Write Operation

- Problem with **test&set**
  - Traffic generated during waiting time
    - Write transaction to the cache block, thus invalidation transactions to other cache holding the block
- Enhancements
  - **test&set** with backoff
  - Test-and-**test&set**
    - Processes busy-wait by repeatedly reading with a standard load (not **test&set**) until the it turns from 1 (locked) to 0 (unlocked)

- 0: free
- 1: locked

Load-Locked, Store-Conditional

- Motivations
  - In addition to spinning with reads rather than read-modify-write (test-and-test&set), we prefer that failed attempts to complete the read-modify-write do not generate invalidations
  - It would be useful to have a single primitive that allows us to implement a range of atomic read-modify-write operations (test&set, fetch&op, compare&swap) rather than implementing each with a separate instruction

Load-Locked (LL), Store-Conditional (SC)

- **Load-locked** (load-linked)
  - Load synchronization variable into a register
  - Followed by arbitrary instructions manipulating the value in the register (modify-part)

- **Store-Conditional**
  - Try to write the register back to memory location if and only if no other processor has written to that location (or cache block) since this processor completed its LL (load-locked)

- 0: free
- 1: locked

```
lock:   ll   reg1, location
        bnz  reg1, lock

        sc   location, reg2
        beqz lock
        ret

unlock: st   location, #0
        ret
```

Atomic Operation Support in RISC-V

- 2 ways
  - LR (Load Reserved) / SC (Store Conditional)
  - AMOs (Atomic Memory Operations)

- Why?
  - There are 2 quite distinct use cases
  - LR/SC
    - Programming language developers assume that the underlying architecture can perform CAS operation
    - CAS is a universal synchronization primitive
    - but requires 3 source operands, complicating instruction format, datapath, control, memory system interface
    - CAS can be implemented with LR/SC
  - AMOs
    - Scale better to large multiprocessor systems than LR/SC
    - Reduction operations can be implemented efficiently
    - Useful for communication with I/O devices
      - Simplify device drivers and improve I/O performance

Source: The RISC-V Reader: An Open Architecture Atlas

LR (Load-Reserved), SC (Store-Conditional) in RISC-V

- **lr.w : lr.w rd, (rs1)**
  - Read a word from the address in rs1
  - Register a *reservation set* - a set of bytes that subsumes the bytes in the addressed word

- **sc.w : sc.w rd, rs2, (rs1)**
  - **`sc.w`** : conditionally [rs1] ← rs2
  - Succeed only if the reservation is still valid and the reservation set contains the bytes being written
  - If it succeeds, [rs1] ← rs2 & rd ← 0
  - If it fails, no write & rd ← none-zero
  - Regardless of success or failure, executing an `sc.w` invalidates any reservation held by this hart

8.2 Load-Reserved/Store-Conditional Instructions

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR.W/D	ordering		0	addr	width	dest	AMO
SC.W/D	ordering		src	addr	width	dest	AMO

```
        // a0: memory location
cas:    lr.w  t0, (a0)        // t0 <= [a0]
        bne   t0, a1, fail    // a1: expected value
        sc.w t0, a2, (a0)     // [a0] <= a2
        bnez t0, cas
        li   a0, 0
        ret

fail:   li    a0, 1
        ret
```

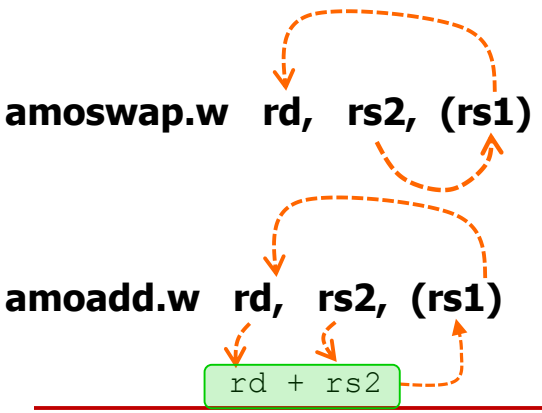
CAS implementation with LR/SC

Atomic Memory Operations (AMO)

- fetch-and-op style atomic primitives

8.4 Atomic Memory Operations

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
AMOSWAP.W/D	ordering		src	addr	width	dest	AMO
AMOADD.W/D	ordering		src	addr	width	dest	AMO
AMOAND.W/D	ordering		src	addr	width	dest	AMO
AMOOR.W/D	ordering		src	addr	width	dest	AMO
AMOXOR.W/D	ordering		src	addr	width	dest	AMO
AMOMAX[U].W/D	ordering		src	addr	width	dest	AMO
AMOMIN[U].W/D	ordering		src	addr	width	dest	AMO



Lock implementation example

```
        li   t0, 1
again:
        lw    t1, (a0)
        bnez  t1, again
        amoswap.w.aq t1, t0 (a0) // attempt to acquire lock
        bnez  t1, again
        #
        # Critical section
        #
        amoswap.w.rl  x0, x0 (a0) // release lock by storing 0
```

Regarding LL / SC

- LL and SC instructions are supported by a number of architectures; Alpah AXP (`ldl_l`/`stl_c`), MIP (`ll`/`sc`) and ARM (`ldrex`/`strex`)
- LL/SC does not suffer from the ABA problem
- In practice, there are often severe restrictions on what a thread can do between a LL and the matching SC
  - A context switch, another LL, or another load or store instruction may cause the SC to fail

Source: The Art of Multiprocessor Programming

LR / SC in RISC-V

- A-extension requires that the address in rs1 be naturally aligned to the size of the operand
  - 4-byte aligned for 32-bit words
  - 8-byte aligned for 64-bit words
  - Otherwise, misaligned exception or access-fault exception generated

- An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword
  - The platform should provide a means to determine the size and shape of the reservation set

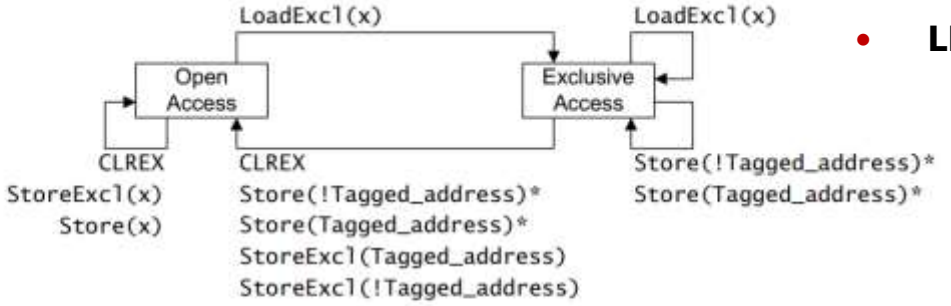
- SC must fail if;
  - a store to the reservation set from another hart occurs between the LR and SC
  - a write from some other device to the bytes accessed by the LR occurs between the LR and SC
  - There is another SC (to any address) between the LR and the SC in program order

Constrained LR/SC loops

The standard A extension defines *constrained LR/SC loops*, which have the following properties:

- The loop comprises only an LR/SC sequence and code to retry the sequence in the case of failure, and must comprise at most 16 instructions placed sequentially in memory.
- An LR/SC sequence begins with an LR instruction and ends with an SC instruction. The dynamic code executed between the LR and SC instructions can only contain instructions from the base "I" instruction set, excluding loads, stores, backward jumps, taken backward branches, JALR, FENCE, FENCE.I, and SYSTEM instructions. If the "C" extension is supported, then compressed forms of the aforementioned "I" instructions are also permitted.
- The code to retry a failing LR/SC sequence can contain backwards jumps and/or branches to repeat the LR/SC sequence, but otherwise has the same constraint as the code between the LR and SC.
- The LR and SC addresses must lie within a memory region with the *LR/SC eventuality* property. The execution environment is responsible for communicating which regions have this property.
- The SC must be to the same effective address and of the same data size as the latest LR executed by the same hart.

Local Monitor in ARMv7 A & R



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instructio
Store represents any other store instruction.

Any LoadExcl operation updates the tagged address to the most significant bits of the address x u

Figure A3-3 Local monitor stat

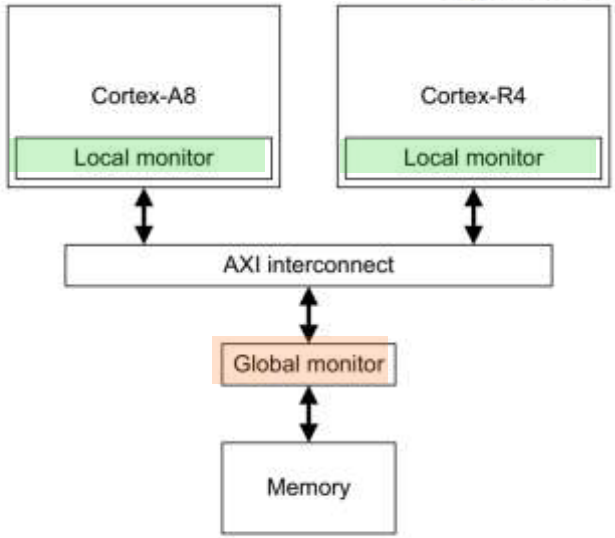


Figure 1-1 Local and global monitors in a multi-core system

- ARMv7 AR Architecture Reference Manual
- ARM Synchronization Primitives (2009)

- **LDREX, STREX and CLREX instructions**

Table A3-3 Effect of Exclusive instructions and write operations on the local monitor

Initial state	Operation[a]	Effect	Final state
Open Access	CLREX	No effect	Open Access
	StoreExcl(x)	Does not update memory, returns status 1	Open Access
	LoadExcl(x)	Loads value from memory, tags address x	Exclusive Access
	Store(x)	Updates memory, no effect on monitor	Open Access
Exclusive Access	CLREX	Clears tagged address	Open Access
	StoreExcl(t)	Updates memory, returns status 0	Open Access
	StoreExcl(!t)	Updates memory, returns status 0[b] / Does not update memory, returns status 1[b]	Open Access
	LoadExcl(x)	Loads value from memory, changes tag to address x	Exclusive Access
Exclusive Access	Store(!t)	Updates memory	Exclusive Access[b] / Open Access[b]
	Store(t)	Updates memory	Exclusive Access[b] / Open Access[b]

a. In the table:
LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any store operation other than a Store-Exclusive operation.
t is the tagged address, bits[31:a] of the address of the last Load-Exclusive instruction. For more information, see *Tagging and the size of the tagged memory block* on page A3-121.

b. IMPLEMENTATION DEFINED alternative actions.

Cortex-A9 in Zynq-7000

- Both data cache read misses and write misses are non-blocking with up to four outstanding data cache read misses and up to four outstanding data cache write misses being supported.
- The APU data caches offer full snoop coherency control utilizing the MESI algorithm.
- The data cache in Cortex-A9 contains local load/store exclusive monitor for LDREX/STREX synchronizations. These instructions are used to implement semaphores. The exclusive monitor handles one address only, with eight 8 words or one cache line granularity. Therefore, avoid interleaving LDREX/STREX sequences and always execute a CLREX instruction as part of any context switch.

Backup Slides

RV32I Base Instruction Set

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
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
0100000	shamt	rs1	101	rd	0010011	SRAI
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
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

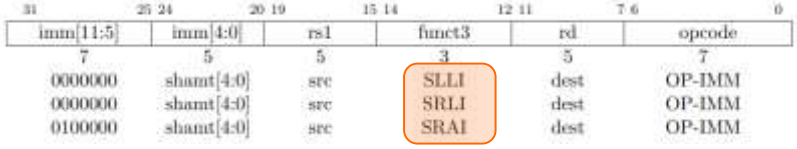
U-type

J-type

B-type

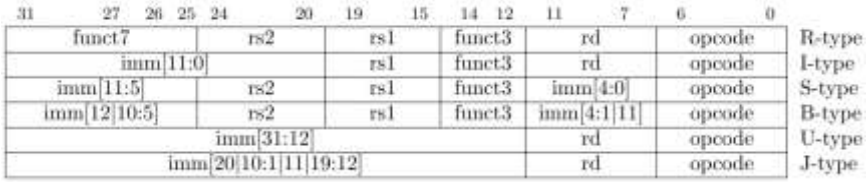
S-type

I-type



I-type specialization

R-type



Korea Univ

Branch Pseudo-instructions

pseudoinstruction	Base Instruction	Meaning
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned

Table 25.2: RISC-V pseudoinstructions.

Condition Field

Every instruction contains a 4-bit condition code field in bits 31 to 28:

Table A3-1 Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See *Condition code 0b1111*	-