Computer Architecture and Organization
Lecture 5

Instruction Set Architecture (ISA)

Learning objective:

You can write programs in assembly for the ARC processor

Topics of this lecture

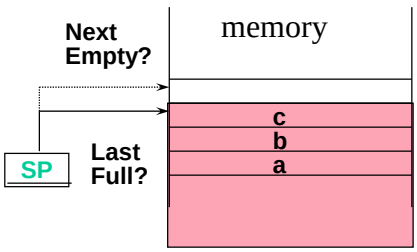
- **Stack**
- **ISA (RISC/CISC)**
- **ARC, A RISC Processor**
  - **ISA**
  - **Pseudo-operations**
  - **Synthetic Instructions**
  - **Examples of Assembly Language Programs**
  - **Subroutine Linkage and Stacks**

Stack

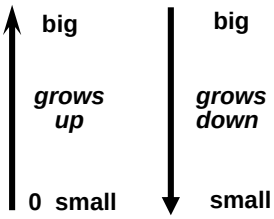
- Stack/LIFO (Last In First Out): memory organization to store and recall elements
  - only top element can be referenced
- Two instructions on top location:
  - POP get top location from the stack
  - PUSH put an element on the top location of the stack
- Processors have (often) a special register:
  - SP (Stack Pointer)
- Two issues for the design of stack:
  - In what direction does the stack grow in the memory?
  - where does the stack-pointer point to?

Stack (Cont')

1) Where does the Stack Pointer point to?



2) How does the stack grow?



Small → Big / SP → Last Full

POP: Reg ← Mem[SP]
SP ← SP - 1

PUSH: SP ← SP + 1
Mem[SP] ← Reg

Small → Big / SP → Next Empty

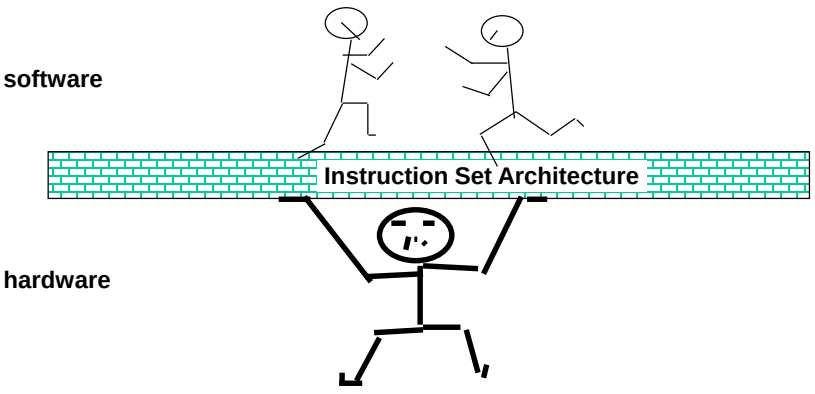
POP: SP ← SP - 1
Reg ← Mem[SP]

PUSH: Mem[SP] ← Reg
SP ← SP + 1

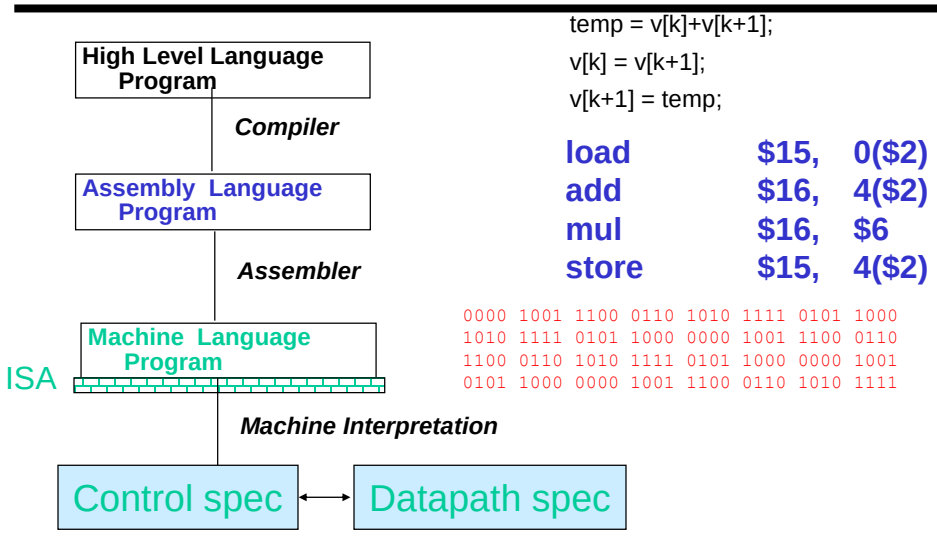
Topics of this lecture

- Stack
- ISA (RISC/CISC) ⇦
- ARC, A RISC Processor
  - ISA
  - Pseudo-operations
  - Synthetic Instructions
  - Examples of Assembly Language Programs
  - Subroutine Linkage and Stacks

The Instruction Set Architecture: the Interface



Levels of abstraction

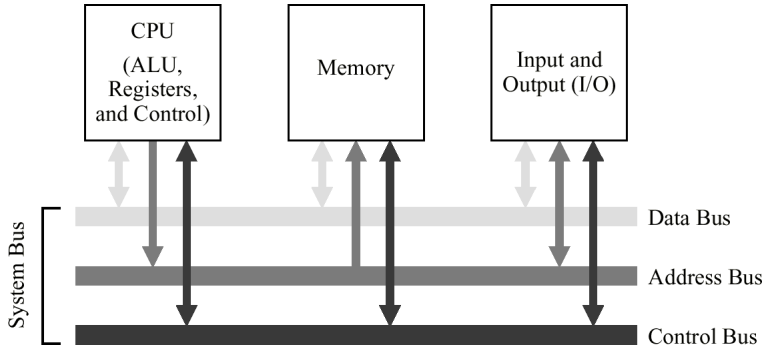


7

7

The System Bus Model of a Computer System

A compiled program is copied from a hard disk to the memory. The CPU reads instructions and data from the memory, executes the instructions, and stores the results back into the memory.

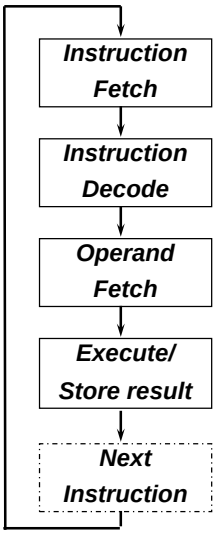


Memory mapped I/O; I/O device uses part of the memory address space

8

8

Instruction Fetch Execute cycle, *also called* Instruction Fetch Decode Execute cycle



- *next instruction is implicit!*

Instructions

The vocabulary of the computer

- **issues:**
  - **instruction format**
  - **location of operands**
  - **number of operands**
  - **addressing modes**
  - **side effects: e.g. condition codes (Zero, Overflow, Negative,..)**

- **Example instruction (ARC processor)**

  add %r1, %r2, %r3 **assembly**

  10 00011 000000 00001 0 00000000 00010 **machine code**

  Often an RTL (Register Transfer Level) notation is used that describes the behavior: %r3 ←%r1+%r2

Note: RTL is not standardized; e.g. another processor company can interpret the assembly instruction as: %r1 ←%r2 + %r3. If not explicitly mentioned else we will use the RTL description of the ARC processor.

Location of operands: General Purpose Registers

- **Since 1975 machines use general purpose registers**
- **Advantages of these registers**
  - **registers are faster than memory (1 clock period vs multiple clock periods)**
  - **registers are easier for a compiler to use**
  - **registers can hold variables**
    - **memory traffic is reduced, so program is speed up**
      - since registers are faster than memory
      - registers can be accessed in parallel
  - **code density improves (since register named with fewer bits than memory location)**

Number of operands: 3 classes

2/3 address machines (can be memory/memory or register/register or register/memory):

2 address	**add A,B**	mem[B] = mem[A] + mem[B]	
3 address	**add A,B,C**	mem[C] = mem[A] + mem[B]	
3 address	**add Ra,Rb,Rc**	Rc = Ra + Rb	

1 address machines (Accumulator machines):

1 address	**add A**	acc = acc + mem[A]	Acc is an implicit register
1 address	**load Rb**	acc = mem[Rb]	
1 address	**store Rb**	mem[Rb] = acc	

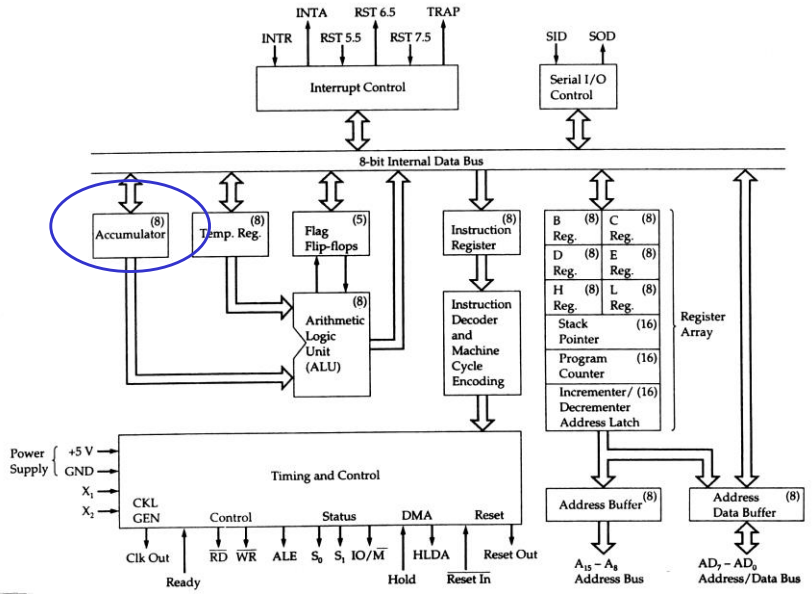
0 address machines (stack):

0 address	**add**	tos = tos + next	(tos=top of stack)

Comparison:

Bytes per instruction? Number of Instructions? Cycles per instruction?

Block diagram Intel 8085



Comparing Number of Instructions

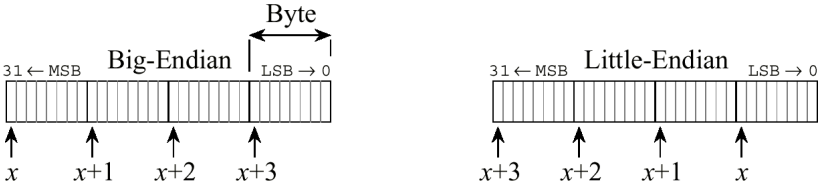
Code sequence for (C = A + B) for four classes of instruction sets (A,B,C are memory addresses)

0 address Stack	1 address Accumulator	2 address (register-memory)	2/3 address (registers)
Push A	Load A	Load A,R1	Load A,R1
Push B	Add B	Add B,R1	Load B,R2
Add	Store C	Store R1,C	Add R1,R2,R3
Pop C			Store R3,C

Note: A,B and C are addresses (long)
R1,R2 and R3 are registers (short)

Addressing Objects: Endianess

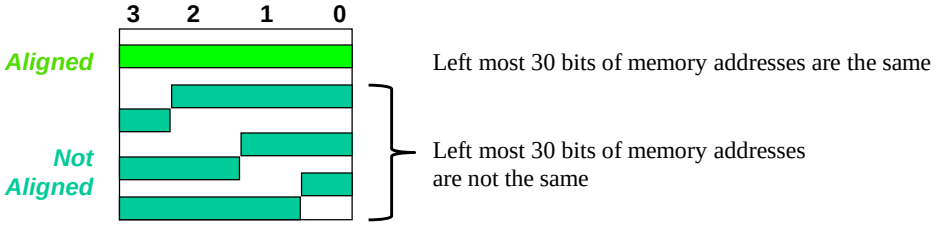
- **Big Endian:** address of most significant byte at lowest byte address
  (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA, Intel-Itanium, ARC, SPARC
- **Little Endian:** address of least significant byte at lowest byte address
  - Intel 80x86, DEC Vax, DEC Alpha, Intel-Itanium



Word address is x for both big-endian and little-endian formats.

Addressing Objects: Alignment

- **Alignment: objects fall on address that is multiple of their size.**
- **E.g. for 32 bits and a memory location stores 1 byte**



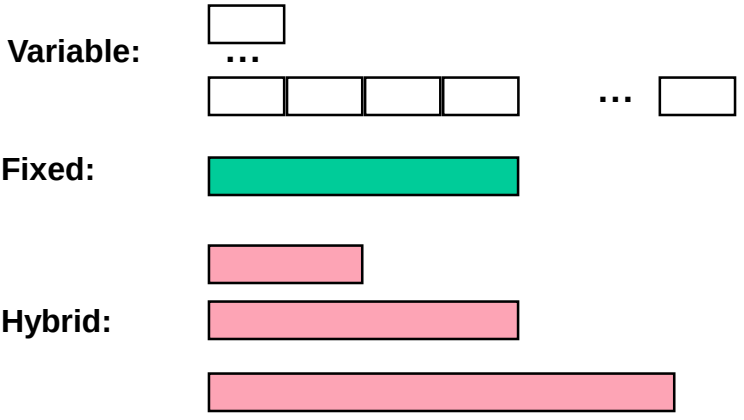
Examples of addressing modes

Addressing mode	example	RTL
Register	Add R3, R4	R4 ← R4+R3
Immediate	Add #3, R4	R4 ← R4 + 3
Register indirect	Add (R3), R4	R4 ← R4 + mem[R3]
Index (displacement)	Add 100[R1], R4	R4 ← R4 + mem[R1 + 100]
Relative	Add $100, R4	R4 ← R4 + mem[PC + 100]
Direct or Absolute	Add 1001, R4	R4 ← R4 + mem[1001]
Memory indirect	Add [1001], R4	R4 ← R4 + mem[mem[1001]]
Post-increment	Add [R1]+, R4	R4 ← R4 + mem[R1] R1 ← R1 + 1
Pre-decrement	Add -[R1], R4	R1 ← R1 - 1 R4 ← R4 + mem[R1]

Many different adressing modes!
Not all ISA's support these modes and/or use different names for it.
Also the syntax of an addressing mode can be different for an ISA.
You do not have to learn the addressing modes on this slide.

17

Instruction Format (1)



19

Instruction Format (2)

- If code size is most important,
  use variable length instructions.

- If performance is most important,
  use fixed length instructions.

RISC/CISC
- CISC (complex instruction set computer)
  Many instruction and/or many addressing modes (e.g. Pentium)
- RISC (reduced instruction set computer)
  - relatively limited number of instructions
  - simpler instructions
  - simple addressing modes (bans indirect addressing)
  (e.g. ARM, MIPS)

Topics of this lecture

- Stack
- ISA (RISC/CISC)
- ARC, A RISC Processor ⇦
  - ISA
  - Pseudo-operations
  - Synthetic Instructions
  - Examples of Assembly Language Programs
  - Subroutine Linkage and Stacks

ARC processor
(based on SPARC processor)

22

Our example processor is the ARC

- The ARC instruction size is 32 bits
- Since 1980 almost every machine uses addresses to level of 8-bits (byte)
- A 32-bit word can be read as four loads of bytes from sequential byte addresses or as one 32-bit word
  - Aligned addressing!
  - Big endian (MSB on lowest address)

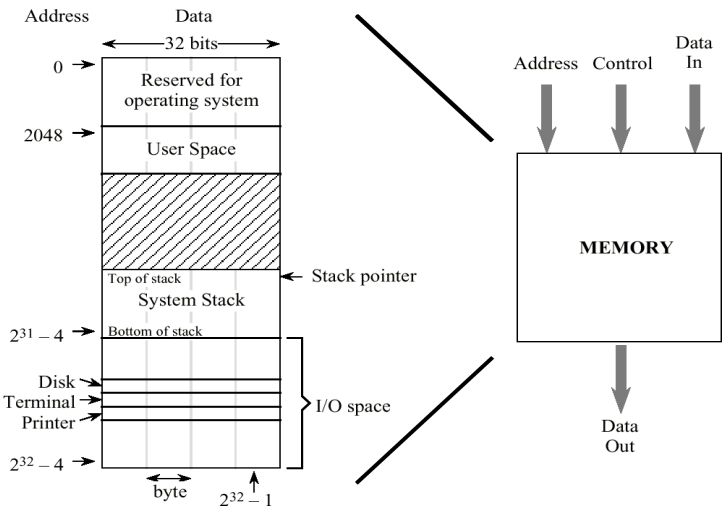
23

Memory Map for the ARC

Memory locations are arranged linearly in consecutive order.

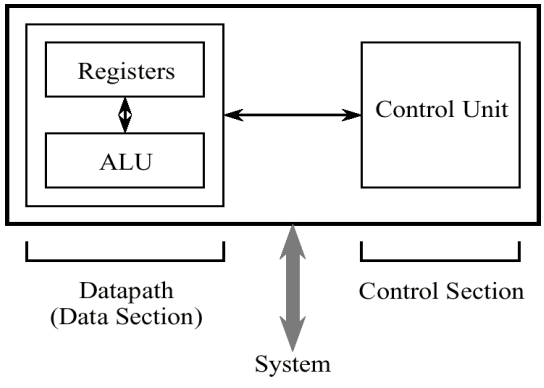
Each numbered location corresponds to an ARC word.

The unique number that identifies each word is referred to as its *address*.

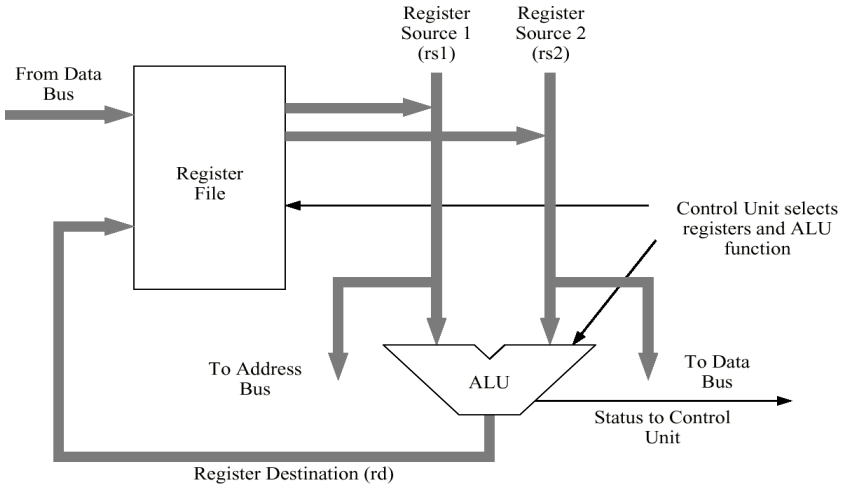


Abstract View of a CPU

The CPU consists of a data section containing registers and an ALU, and a control section, which interprets instructions and effects register transfers. The data section is also known as the *datapath*.



Simplified view of datapath ARC processor



The ARC datapath is made up of a collection of registers known as the *register file* and the *arithmetic and logic unit* (ALU).

26

ARC User-Visible Registers

Register 00	%r0 [= 0]	Register 11	%r11		Register 22	%r22		
Register 01	%r1	Register 12	%r12		Register 23	%r23		
Register 02	%r2	Register 13	%r13		Register 24	%r24		
Register 03	%r3	Register14	%r14	[%sp]	Register 25	%r25		
Register 04	%r4	Register 15	%r15	[link]	Register 26	%r26		
Register 05	%r5	Register 16	%r16		Register 27	%r27		
Register 06	%r6	Register 17	%r17		Register 28	%r28		
Register 07	%r7	Register 18	%r18		Register 29	%r29		
Register 08	%r8	Register 19	%r19		Register 30	%r30		
Register 09	%r9	Register 20	%r20		Register 31	%r31		
Register 10	%r10	Register 21	%r21					

%r0 is always 0

PSR	%psr

32 bits

PC	%pc

32 bits

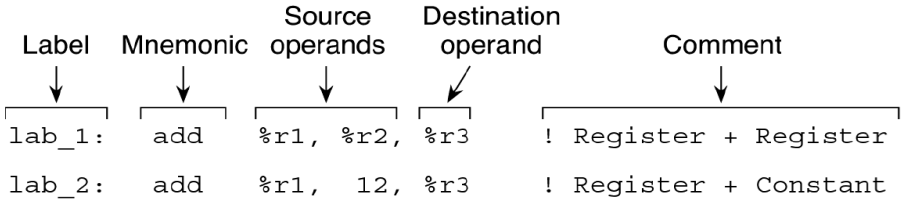
- **'visible' means part of the ISA**

psr: processor status register
sp: stack pointer
pc: program counter
link: return address of function call

27

ARC Assembly Language Format

- The ARC assembly language format is the same as the SPARC assembly language format.
- This example shows the assembly language format for ARC (and SPARC) arithmetic and logic instructions.

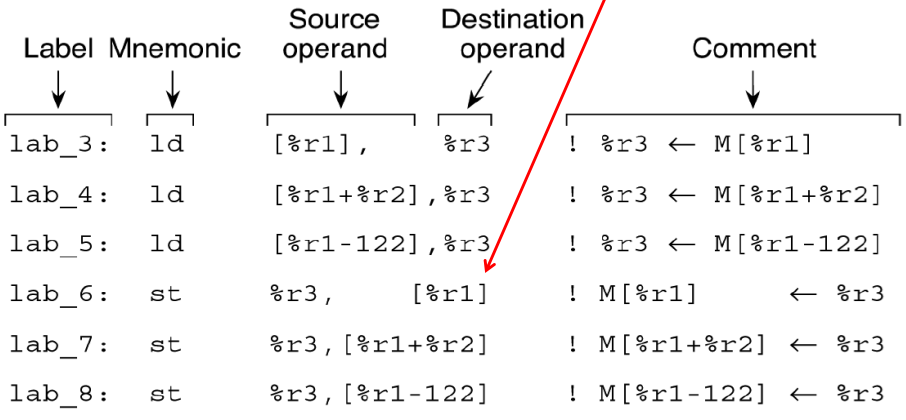


28

ARC Load / Store Format

In ARCTool: st %r3, %r1 (no brackets)

- This example shows the assembly language format for ARC load and store instructions.



29

Simple Example: Add Two Numbers

- The figure shows a simple program fragment using our `ld`, `st`, and `add` instructions. This fragment is equivalent to the C statement:

```
z = x + y;
```

```
ld      [x],    %r1
ld      [y],    %r2
add     %r1,    %r2,    %r3
st      %r3     [z]
```

- The ARC must first fetch the **`x`** and **`y`** operands from memory using LD instructions, and then perform the addition, and then store the result back into **`z`** using an ST instruction.

file: add2numbers.asm (on Canvas)

ARC Fragment that Computes the Absolute Value

The function `abs` realized on the ARC processor:

```
abs(x) := if (x < 0) then x = -x;
```

the ARC fragment to implement this is:

```
.begin
.org 0
        ld[x], %r1
        subcc %r0, %r1, %r1
        bneg over
        st %r1, [x]
over:   halt
x:      10
.end
```

RTL: %r1←%r0-%r1
bneg: branch if negative (N bit)

Remember: %r0 is always 0

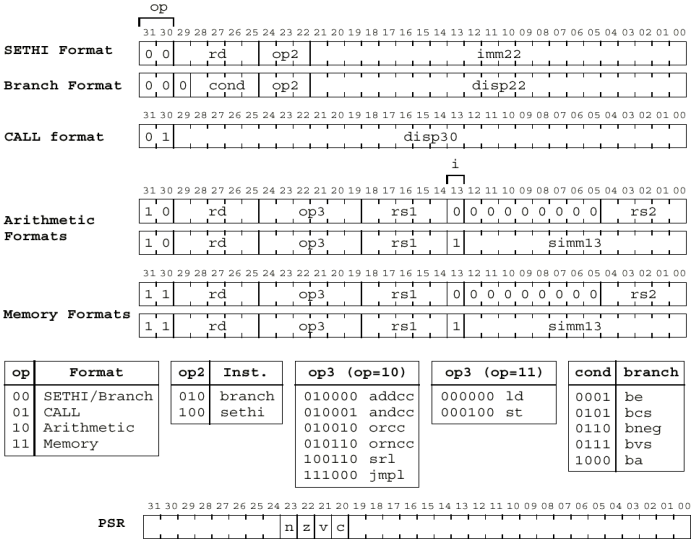
File: abs.asm

A Portion of the ARC ISA

- The ARC ISA is a subset of the SPARC ISA. A portion of the ARC ISA is shown here (the implementation in chapter 5 supports only this subset).

	Mnemonic	Meaning
Memory	**ld**	Load a register from memory
	st	Store a register into memory
Logic	**sethi**	Load the 22 most significant bits of a register
	andcc	Bitwise logical AND
	orcc	Bitwise logical OR
	orncc	Bitwise logical OR of rs1 and the inverse of rs2
	srl	Shift right (logical)
Arithmetic	**addcc**	Add
Control	**call**	Call subroutine
	jmpl	Jump and link (return from subroutine call)
	be	Branch if equal
	bneg	Branch if negative
	bcs	Branch on carry
	bvs	Branch on overflow
	ba	Branch always

ARC Instruction and PSR Formats



PSR = Processor State Register

%r0 is always 0; %r0 Trick

- **Load a register from main memory:**
  - **The memory address is computed from two components:**
    - **rs1 + rs2, or**
    - **rs1 + simm13 (sign extended).**
  - **If one component is not needed the assembler inserts %r0 (constant 0)**
- **Examples:**

Table 4.2 Load instruction syntax and meaning

Example	Programmer writes:	Assembler inserts	Meaning
a)	`ld [%r5 + x], %r1`	`ld [%r5 + x], %r1`	`%r1 ← M[%r5 + x]`
b)	`ld [%r2 + %r4], %r1`	`ld [%r2 + %r4], %r1`	`%r1 ← M[%r2 + %r4]`
c)	`ld [x], %r1`	`ld [%r0 + x], %r1`	`%r1 ← M[x]`
d)	`ld [%r3], %r1`	`ld [%r3+ %r0], %r1`	`%r1 ← M[%r3]`

34

Examples: ld, sethi

Table 4.3 Example (a): `ld [%r5 + x], %r1` with x is 2064

Field Name:	op	rd	op3 (ld)	rs1	i	simm13
Field Size (bits):	2	5	6	5	1	13
Field ID:		%r1	000000	%r5	1	2064
Object code:	11	00001	000000	00101	1	0100000010000

Table 4.4 Example (b): `ld [%r2 + %r4], %r1`

Field Name:	op	rd	op3 (ld)	rs1	i	00000000	rs2
Field Size (bits):	2	5	6	5	1	8	5
Field ID:		%r1	000000	%r2	0	00000000	%r4
Object code:	11	00001	000000	00010	0	00000000	00100

Table 4.6 Encoding `sethi 0x304F15, %r1`

Field Name:	op	rd	op2	imm22
Field Size:	2	5	3	22
Field ID:	2	%r1	100	0x304F15
Object code:	00	00001	100	1100000100111100010101

35

ARC Pseudo-Ops

Pseudo-Op	Usage	Meaning
.equ X	.equ #10	Treat symbol X as $(10)_{16}$
.begin	.begin	Start assembling
.end	.end	Stop assembling
.org	.org 2048	Change location counter to 2048
.dwb	.dwb 25	Reserve a block of 25 words
.global	.global Y	Y is used in another module
.extern	.extern Z	Z is defined in another module
.macro	.macro M a, b, ...	Define macro M with formal parameters a, b, ...
.endmacro	.endmacro	End of macro definition
.if	.if <cond>	Assemble if <cond> is true
.endif	.endif	End of .if construct
.align	.align 4	Round location counter up to even multiple of 4.

Pseudo-ops are instructions to the assembler. They are not part of the ISA, but instruct the assembler to do an operation at assembly time.

In this course we only use: .equ, .begin, .end, .org

36

Synthetic Instructions

- Many assemblers will accept synthetic instructions that are converted to actual machine-language instructions during assembly. The figure below shows some commonly used synthetic instructions.

Synthetic Instruction	Instruction Generated	Comment
not *rs1*, *rd*	xnor *rs1*, %r0, *rd*	1's complement
neg *rs1*, *rd*	sub %r0, *rs1*, *rd*	2's complement
inc *rd*	add *rd*, 1, *rd*	increment by 1
dec *rd*	sub *rd*, 1, *rd*	decrement by 1
clr *rd*	and *rd*, %r0, *rd*	clear a register
cmp *rs1, reg_or_imm*	subcc *rs1, reg_or_imm, %r0*	compare, set ccs
tst *rs1*	orcc %r0, *rs1*, %r0	test
mov *reg_or_imm*, *rd*	or %r0, *reg_or_imm*, *rd*	Move a value
set value, *rd*	or %r0, value, *rd*	-4096 ≤ value ≤ 4095

A synthetic instruction is replaced with a single instruction
A Macro is replaced with a group of instructions (see chapter 6, page 219).

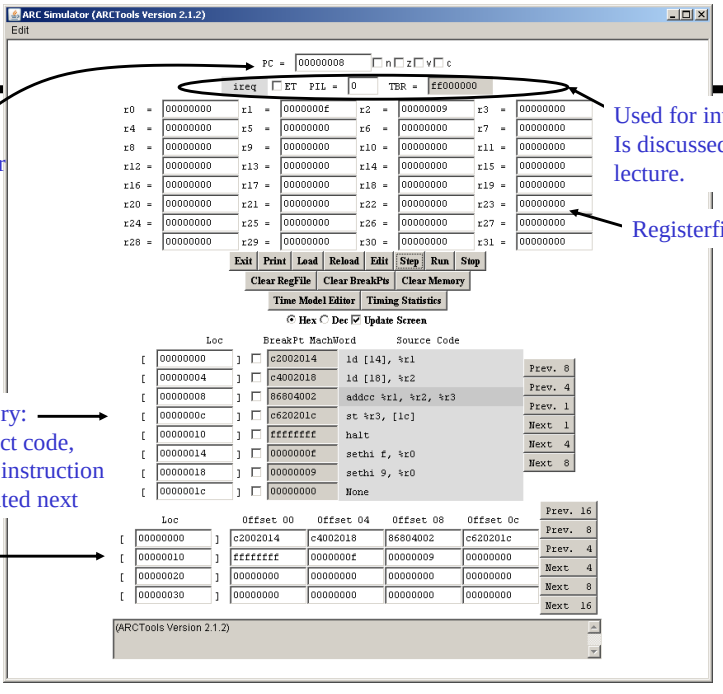
37

ARC Example Program

```
! This program adds two numbers
        .begin
        .org 0
        ld      [x], %r1          ! Load x into %r1
        ld      [y], %r2          ! Load y into %r2
        addcc   %r1, %r2, %r3     ! %r3 <- %r1 + %r2
        st     %r3, [z]           ! store %r3 into z
        halt
x: 15
y: 9
z: 0
        .end
<empty line after ".end">
```


38

Simulator



Program Counter and status bits

Used for interrupts. Is discussed in another lecture.

Registerfile

Program in memory: Source code, object code, highlighted is the instruction that is to be executed next

Main memory

39

DEMO ARCtools

Read Appendix B!
B.1 till B.6

40

A More Complex Example Program

Is the same as:
ld [%r4], %r5

An ARC program sums five integers.

```
! This program sums LENGTH numbers
! Register usage:       %r1 - Length of array a
!                       %r2 - Starting address of array a
!                       %r3 - The partial sum
!                       %r4 - Pointer into array a
!                       %r5 - Holds an element of a
            .begin              ! Start assembling
            .org  2048          ! Start program at 2048
a_start     .equ  3000          ! Address of array a
            ld    [length], %r1 ! %r1 <- length of array a
            ld    [address],%r2 ! %r2 <- address of a
            andcc %r3, %r0, %r3 ! %r3 <- 0
loop:       andcc %r1, %r1, %r0 ! Test # remaining elements
            be    done          ! Finished when length=0
            addcc %r1, -4, %r1  ! Decrement array length
            addcc %r1, %r2, %r4 ! Address of next element
            ld    %r4, %r5      ! %r5 <- Memory[%r4]
            addcc %r3, %r5, %r3 ! Sum new element into r3
            ba    loop          ! Repeat loop.

done:       jmpl  %r15 + 4, %r0 ! Return to calling routine

length:       20                ! 5 numbers (20 bytes) in a
address:      a_start
            .org  a_start       ! Start of array a
a:            25                ! length/4 values follow
             -10
              33
              -5
               7
            .end                    ! Stop assembling
```

On the line .begin comment is not supported by the ARCTool !

jmpl is explained in a moment. For now it can be replaced with halt.

File: fig4_22.asm

41

subroutine

- Most programming languages support subroutines (functions, procedures)
- How to pass the arguments. Three ways are shown
  - Via registers in the register file
  - Via a data link area in main memory
  - Via a stack

42

Subroutine Linkage – Registers

- Subroutine linkage with registers passes parameters in registers.

Call:
Will store its current address in %r15
(by convention) and then jumps to add_1

Jmpl:
A jump is made to address %r15+4.
That is the instruction after the
"call add_1" instruction.

```
! Calling routine
      :
      :
     ld     [x], %r1
     ld     [y], %r2
     call   add_1
     st     %r3, [z]
      :
      :
x:   53
y:   10
z:    0
```

```
! Called routine
! %r3 ← %r1 + %r2


add_1:  addcc   %r1, %r2, %r3
        jmpl    %r15 + 4, %r0
```

Jmpl can store its address in this register
(in this case %r0 is used and will always be 0,
see more on page 120. We will not use it this way).

File: fig4_23_subroutine.asm

43

sethi

- **It is not possible to load a register with a 32 bits value.**
- **Sethi const22,%r1 will load the 22 bits constant value in the upper 22 bits of register r1; lower 10 bits are 0**
- **Suppose you want to store 0x89ABCDEF (0x..→hex) in %r3**
  - **1000 1001 1010 1011 1100 1101 1110 1111**
  - **Top 22 bits in blue → 10 0010 0110 1010 1111 0011→0x226AF3**
  - **Lower 10 bits→ 01 1110 1111 →0x1EF**
  - **Two instructions are needed, e.g.**

    sethi 0x226AF3, %r3

    or %r3, 0x1EF, %r3

44

Subroutine Linkage – Data Link Area

- Subroutine linkage with a data link area passes parameters in a separate area in memory. The address of the memory area is passed in a register (`%r5` here).

```
! Calling routine
     .
     .
     .
    st     %r1, [x]
    st     %r2, [x+4]
    sethi  x, %r5
    srl    %r5, 10, %r5
    call   add_2
    ld     [x+8], %r3
     .
     .
     .
! Data link area
x:  .dwb   3
```

```
! Called routine
! x[2] ← x[0] + x[1]

add_2:  ld      %r5, %r8
        ld      %r5 + 4, %r9
        addcc   %r8, %r9, %r10
        st      %r10, %r5 + 8
        jmpl    %r15 + 4, %r0
```

Note:
ld %r5, %r8 is the same as ld [%r5],%r8
st %r10,%r5+8 is the same as st %r10, [%r5+8]

File: fig4_24_data_link_area.asm

Subroutine Linkage – Stack

- Subroutine linkage with a stack passes parameters on a stack.

```
  ! Calling routine
      .
      .
      .
%sp  .equ   %r14
     addcc  %sp, -4, %sp
     st     %r1, %sp
     addcc  %sp, -4, %sp
     st     %r2, %sp
     call   add_3
     ld     %sp, %r3
     addcc  %sp, 4, %sp
      .
      .
      .
```

```
! Called routine
! Arguments are on stack.
! %sp[0] ← %sp[0] + %sp[4]
    %sp  .equ   %r14
add_3:  ld      %sp, %r8
        addcc   %sp, 4, %sp
        ld      %sp, %r9
        addcc   %r8, %r9, %r10
        st      %r10, %sp
        jmpl    %r15 + 4, %r0
```

addcc %sp, 4, %sp

addcc %sp, - 4, %sp

Optimized program: the red %sp add and subtract intructions removed

%sp .equ %r14
is not supported by ARCTools. I replaced it with:
addcc %r0, 128, %r14
(stack pointer points to address 128)

Replace occurrences of %sp with %r14

File: fig4_25_subr_stack.asm (next slide)

46

Complete example with stack

```
.begin
.org 0
addcc %r0, 128, %r14 ! init SP (%r14)
ld [op1], %r1
ld [op2], %r2
addcc %r14, -4, %r14
st %r1, %r14
addcc %r14, -4, %r14
st %r2, %r14
call add_3
ld %r14, %r3
addcc %r14, 4, %r14
st %r3, [op3]
halt
```

```
! Called routine
! Arguments are on stack
! %sp[0] <- %sp[0] + %sp[4]
add_3:  ld %r14, %r8
        addcc %r14, 4, %r14
        ld %r14, %r9
        addcc %r14, 4, %r14
        addcc %r8, %r9, %r10
        addcc %r14, -4, %r14
        st %r10, %r14
        jmpl %r15+4, %r0

! data
op1:    53
op2:    10
op3:    0
.end
```

SP incremented and decremented. Optimize: remove both instructions

47

Next lecture

- Data path & Controller (subset) ARC
- Read chapter 5 and don't forget the exercises!

Download from

http://iiusatech.com/murdocca/CAO

- **ARCTools**
  (use: Download ARCTools (iiusatech.com http mirror))

48