# Lecture 7.01
# Assembly Process – 1

**CPS310**

**Computer Organization II**

**WINTER 2022**

# Review

- Instruction Set Architecture (ISA)
- Byte-Addressable Machine
- Big-Endian vs Little-Endian
- RISC vs CISC
- ARC (A RISC Computer)
- ARC Memory Map
- ARC Datapath
- ARC ISA Categories
- PSR Condition Codes (Z, N, C, V)
- ARC Assembly Language Format
- ARC Assembly Programming
- Addressing mode

# The Assembly Process

- The process of translating an assembly language program into a machine language program

- Generally provide support:
  - Allow programmer to specify locations of data and code
  - Translate valid assembly language statements into the equivalent machine language

# The Assembly Process

- Let the programmer to specify the starting address of the program

- Permit symbolic labels to represent addresses and constants

- Allow variables to be defined in one assembly language program and used in another, separately assembled program

- Support macro expansion

# 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 |

***- instructions to the assembler to perform some action at assembly time***
***- Not part of the ISA***

# Pseudo-Ops vs Instructions

- Instructions are specific to a given machine

- Pseudo-Ops are specific to a given assembler

Most commonly used Pseudo-Ops:

**.equu**: tell the assembler to equate a value or a string to a symbol

**.begin/.end**: tell the assembler where to start/stop the assembling process

**.org**: tell the assembler the address of the next instruction

# Pseudo-Ops

**.global**: makes a label available for use in other modules

**.extern:** identifies a label that is defined in another modules

- Used in Linking and Loading

①

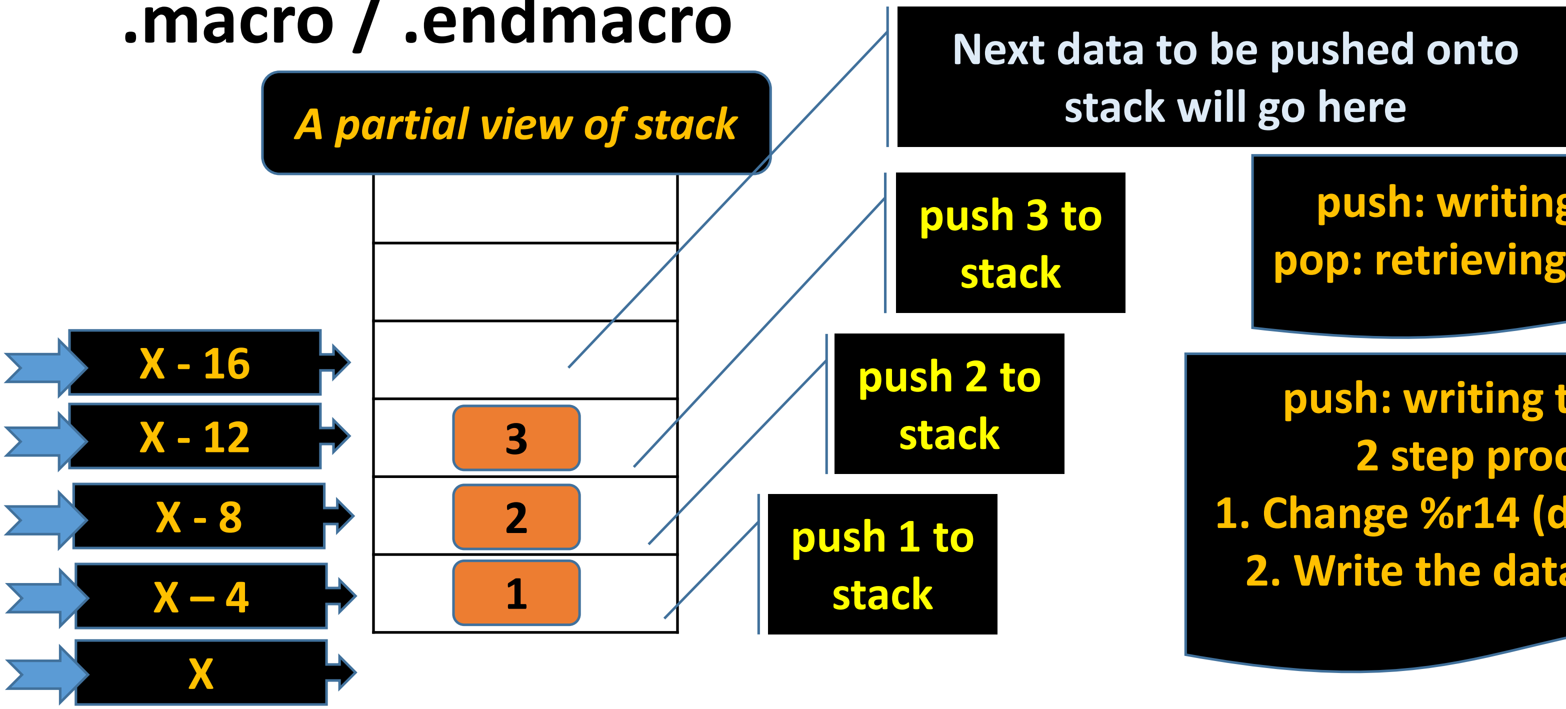
```
! Main Program
        .begin
        .org     2048
        .extern  subr
Main:   ld       [x], %r2
        ld       [y], %r3
        call     subr
        jmpl     %r15+4, %r0
X:      105
Y:      92
        .end
```

②

```
!  Subroutine Library
        .begin
ONE     .equ     1
        .org     2048
        .global  subr
subr:   orncc    %r3, %r0, %r3
        addcc    %r3, ONE, %r3
        jmpl     %r15+4, %r0
        .end
```

# Pseudo-Ops

## .macro / .endmacro

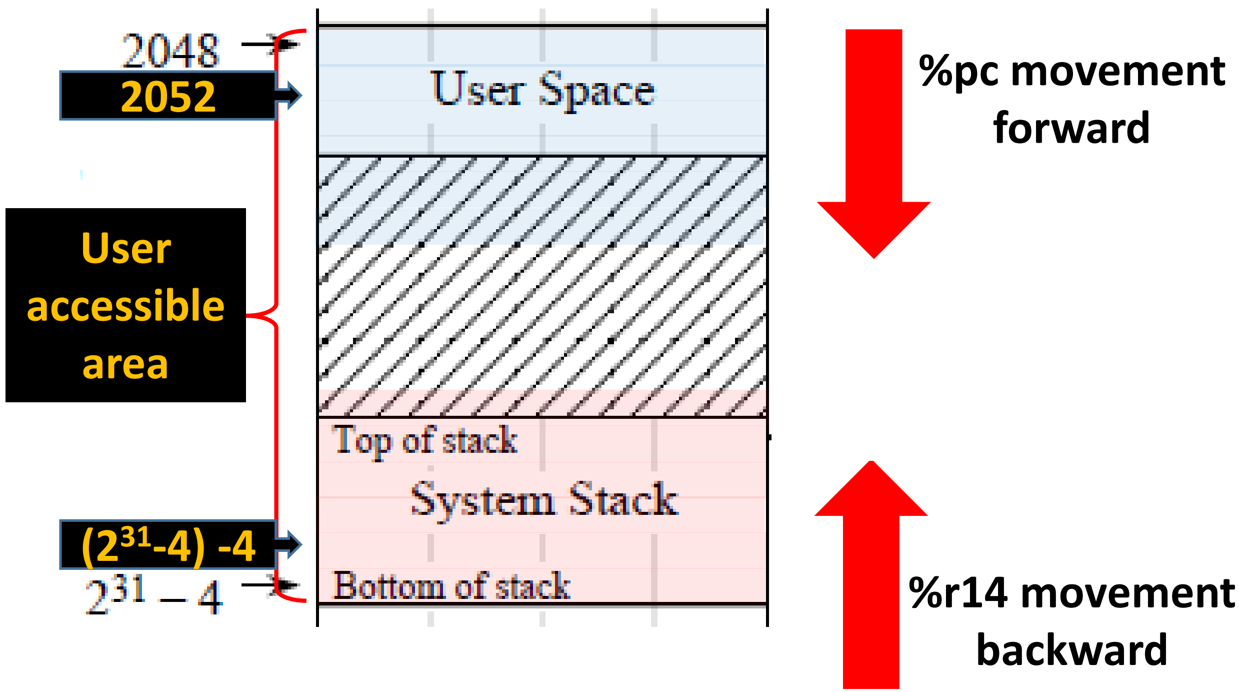



push: writing to stack
pop: retrieving from stack

push: writing to stack
2 step process
1. Change %r14 (decrement)
2. Write the data to stack

```
! Macro definition for 'push'
.macro   push    arg1                    ! Start macro definition
addcc            %r14, -4, %r14          ! Decrement SP
st               arg1, %r14              ! Push arg1 onto stack
.endmacro
```

# Stack Pointer (%r14) in ARC




%PC is pointing to 2048

What is next value of %PC?

How does %PC change?

%PC is pointing to 2048+4

%r14 is pointing to $2^{31}$-4

What is next value of %r14?

How does %r14 change?

%r14 is pointing to ($2^{31}$-4)-4

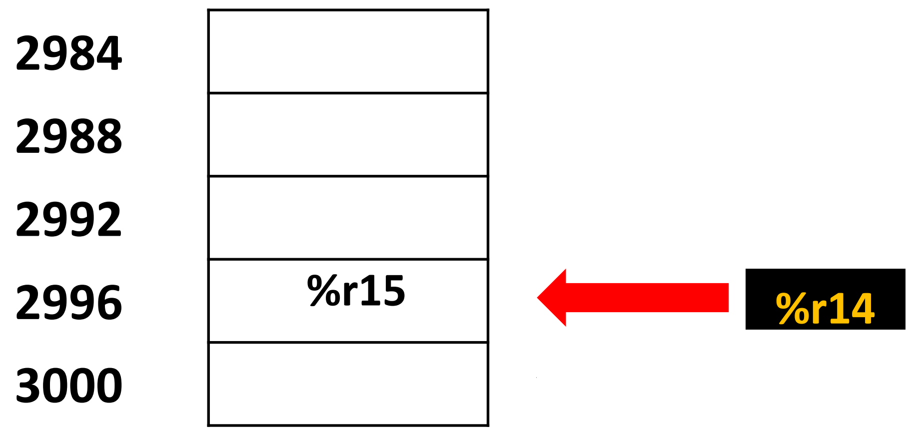
# Pseudo-Ops

## .macro / .endmacro

```
! Macro definition for 'push'
.macro    push    arg1                   ! Start macro definition
addcc             %r14, -4, %r14         ! Decrement SP
st                arg1, %r14             ! Push arg1 onto stack
.endmacro
```

Using the defined macro

```
push    %r15    ! Push %r15 onto stack, assuming %r14 is pointing to 3000
```



# ARC Example Program

**An ARC assembly language program to add two integers:**

```
! This programs adds two numbers

        .begin
        .org 2048
prog1:  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
        jmpl    %r15 + 4, %r0     ! Return
x:      15
y:      9
z:      0
        .end
```

# Partial View of the Memory Map

| Address | Memory Content (non-binary view) | |
|---|---|---|
| 2048 – prog1 | ld | [x], %r1 |
| 2052 | ld | [y], %r2 |
| 2056 | addcc | %r1,%r2, %r3 |
| 2060 | st | %r3, [z] |
| 2064 | jmpl | %r15+4, %r0 |
| 2068 – x | 15 | |
| 2072 – y | 9 | |
| 2076 – z | 0 | |

```
!               %r1:    length of array a
!               %r2:    starting address of array a
!               %r3:    the partial sum
!               %r4:    pointer to array a
!               %r5:    holds an element of a
```

```
.begin
                .org    2048
a_start         .equ    3000
                ld [length], %r1
                ld [address], %r2
                andcc %r3, %r0, %r3
loop:           andcc %r1, %r1, %r0
                be done
                addcc %r1, -4, %r1
                addcc %r1, %r2, %r4
                ld %r4, %r5
                addcc %r3, %r5, %r3
                ba loop
done:           jmpl %r15 + 4, %r0
```

```
length:         20
address:        a_start
                .org a_start
a:              25
                -10
                3
                -5
                7
                .end
```

# Partial View of the Memory Map

| | | |
|---|---|---|
| 2048 | ld [length], %r1 | ! %r1 ← Length of array a |
| 2052 | ld [address], %r2 | ! %r2 ← Address of a |
| 2056 | andcc %r3, %r0, %r3 | ! %r3 ← 0 |
| 2060 – loop | andcc %r1, %r1, %r0 | ! Is %r1 0? |
| 2064 | be done | ! If 0 then branch to location done |
| 2068 | addcc %r1, -4, %r1 | ! %r1 ← decrement length of array a |
| 2072 | addcc %r1, %r2, %r4 | ! %r4 ← address of next element of a |
| 2076 | ld %r4, %r5 | ! %r5 ← next element |
| 2080 | addcc %r3, %r5, %r3 | ! %r3 ← %r3 + next element |
| 2084 | ba loop | ! Always branch to location loop |
| 2088 – done | jmpl %r15 + 4, %r0 | |
| 2092 – length | 20 | ! Length of array a |
| 2096 – address | 3000 | ! Starting address of a |
| 3000 – a | 25 | ! a[0] |
| 3004 | -10 | ! a[1] |
| 3008 | 33 | ! a[2] |
| 3012 | -5 | ! a[3] |
| 3016 | 7 | ! a[4] |

# A SUBSET OF INSTRUCTION SETS FOR THE ARC ISA

| | 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 NOR |
| | **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 Sets Supported by most Simulators

- nop
- **sethi**
- **be, bcs, bneg, bvs, ba,** bne, bcc, bpos, bvc
- **call**
- **jmpl**
- **addcc, andcc,** subcc**, orcc, orncc, xorcc**
- **srl,** sll, sra
- add, sub, and, or, orn, xor
- **ld, st**

**Note: some assemblers might only support a subset of these instructions.**

# Branch Instructions

**be**: *branch on Zero*

**If the Z condition code is 1,**
**then branch to the address represented by the label which is the instruction operand.**

**bneg**: *branch on Negative*

**If the N condition code is 1,**
**then branch to the address represented by the label which is the instruction operand.**

**bcs**: *branch on Carry*

**If the C condition code is 1,**
**then branch to the address represented by the label which is the instruction operand.**

# Other Branch Instructions

**bvs**: *branch on oVerflow*

If the V condition code is 1,
then branch to the address represented by the label which is the instruction operand.

**ba**: *branch always*

Always branch to the address represented by the label which is the instruction operand.

# Other Branch Instructions

**bpos**: *branch on Positive*

If the condition codes signal a positive result, then branch to the address represented by the label which is the instruction operand.

**Example:**

```
bpos label          !Branch if positive
```

optional

**bcc**: *branch on Carry Clear (not carry)*

If the c condition code is 0, then branch to the address represented by the label which is the instruction operand.

**Example:**

```
bcc label           !Branch to label if C is 0
```

# Other Branch Instructions

**bvc**: *branch on No oVerflow*
**If the v condition code is 0, then branch to the address represented by the label which is the instruction operand.**

optional

**Example:**

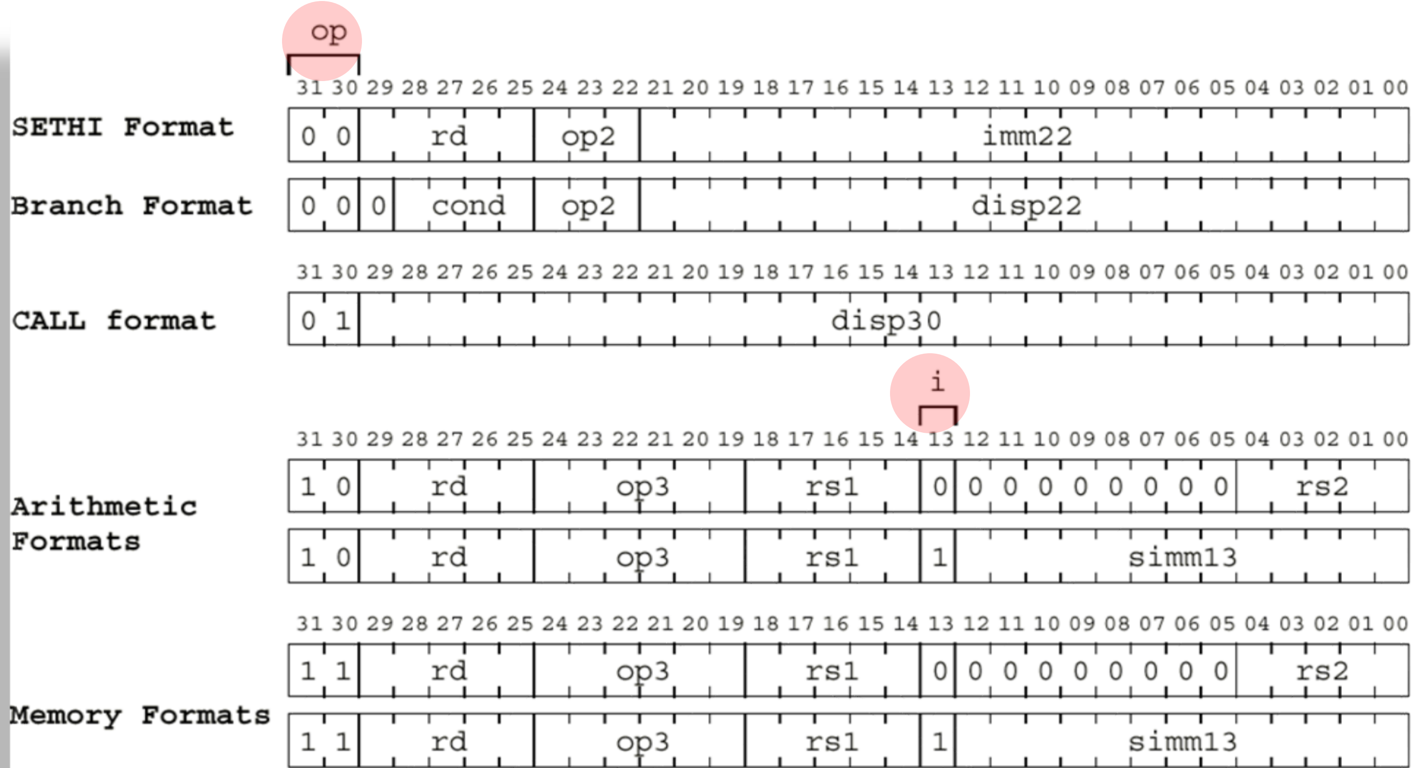
```
bvs label        !Branch to label if V is 0
```

**bne**: *branch on not Zero*
**Branch if not equal to zero to the address represented by the label which is the instruction operand.**

**Example:**

```
bne label        !Branch to label if not equal to zero
```

# Instruction Formats and PSR Format for the ARC



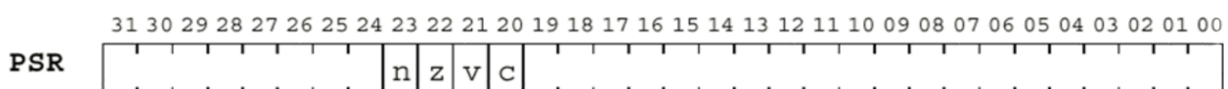

| op | Format |
|---|---|
| 00 | SETHI/Branch |
| 01 | CALL |
| 10 | Arithmetic |
| 11 | Memory |

| op2 | Inst. |
|---|---|
| 010 | branch |
| 100 | sethi |

| op3 (op=10) | |
|---|---|
| 010000 | addcc |
| 010001 | andcc |
| 010010 | orcc |
| 010110 | orncc |
| 100110 | srl |
| 111000 | jmpl |

| op3 (op=11) | |
|---|---|
| 000000 | ld |
| 000100 | st |

| cond | branch |
|---|---|
| 0001 | be |
| 0101 | bcs |
| 0110 | bneg |
| 0111 | bvs |
| 1000 | ba |



# Understanding the bits patterns

| 31 30 | 29 28 27 26 25 | 24 23 22 21 20 19 | 18 17 16 15 14 | 13 | 12 11 10 09 08 07 06 05 | 04 03 02 01 00 |
|---|---|---|---|---|---|---|
| 1 1 | rd | op3 | rs1 | 0 | 0 0 0 0 0 0 0 0 | rs2 |
| 1 1 | rd | op3 | rs1 | 1 | simm13 | |

**31, 30**: 2 bits to recognize the operation type (**op**)

**29-25**: 5 bits to recognize **rd**

**24-19**: 6 bits to recognize **op3**

**18-14**: 5 bits to recognize **rs1**

**13**: 1 bit hardcoded to indicate if mem is being used or not (**i**)

**12-0**: 13 bits for simm13

**04-0**: 5 bits for **rs2**

# LOAD (ld)

- **Load a register from the memory**
- Memory address must be aligned on a word boundary
- How to compute the address?
  **(1) rs1 + rs2**
  **(2) rs1 + simm13**
- Example: Copy content of location **x** to **%r1**

**ld [x], %r1**

- All these instruction do the same thing
- **ld [x], %r0, %r1 !(***)**
- **ld %r0+x, %r1**

# ld [x], %r1

given

- Let’s assume **x** is $2064)_{10}$

From Instruction Format Table

| op3 (op=11) | |
|---|---|
| 000000 | ld |
| 000100 | st |

| 31 30 | 29 28 27 26 25 | 24 23 22 21 20 19 | 18 17 16 15 14 | 13 | 12 11 10 09 08 07 06 05 | 04 03 02 01 00 |
|---|---|---|---|---|---|---|
| 1 1 | rd | op3 | rs1 | 0 | 0 0 0 0 0 0 0 0 | rs2 |
| 1 1 | rd | op3 | rs1 | 1 | simm13 | |

$2064)_{10}$ = 0 1000 0001 0000

| 11 | rd | op3 | rs1 | 1 | simm13 |
|---|---|---|---|---|---|
| 11 | 00001 | 000000 | 00000 | 1 | 0 1000 0001 0000 |
| | %r1 | | %r0 | | |

# STORE (st)

- **Store a register into memory**
- Memory address must be aligned on a word boundary
- How to compute the address?

  **(1) rs1 + rs2**

  **(2) rs1 + simm13 (A combination of rs & simm13 provides the address)**

  **(3) rd provides the source**

- Example: Store content of **r1** in memory location **x**

```
st   %r1, [x]          ! x is 2064)10
```

# st %r1, [x]

- **x** is $2064)_{10}$

From Instruction Format Table

| op3 (op=11) | |
|---|---|
| 000000 | ld |
| 000100 | st |

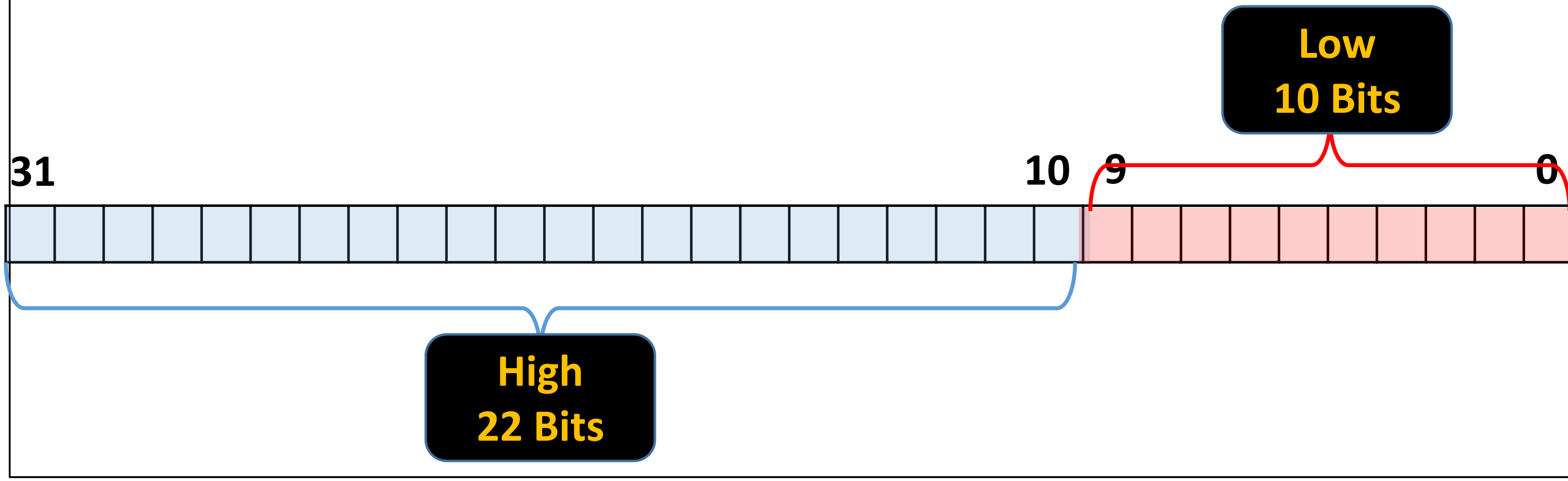
| 31 | 30 | 29–25 | 24–19 | 18–14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04–00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | rd | op3 | rs1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 |
| 1 | 1 | rd | op3 | rs1 | 1 | simm13 | | | | | | | | |

$2064)_{10}$ = 0 1000 0001 0000

| 11 | rd | op3 | rs1 | 1 | simm13 |
|---|---|---|---|---|---|
| 11 | 00001 | 000100 | 00000 | 1 | 0 1000 0001 0000 |
| | %r1 | | %r0 | | |

# sethi

- For a register set the high 22 bits
- The low 10 bits go to ZERO
- If the register is **%r0**, then **NOP**

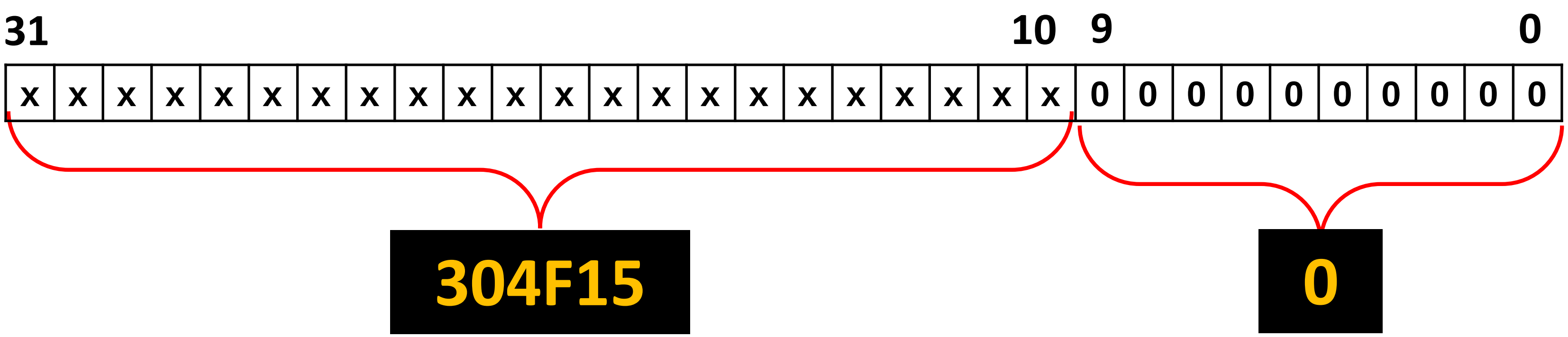


# sethi 0x304F15, %r1

**In register %r1:**

**Set the high 22 bits to** **0x304F15**

**Set the low 10 bits go to** **0**



# sethi 0x304F15, %r1

**Set the high 22 bits of %r1 to 0x304F15 and set the lower 10 bits to zero**

| op2 | Inst. |
|---|---|
| 010 | branch |
| 100 | sethi |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | rd | | | | | op2 | | | imm22 | | | | | | | | | | | | | | | | | | | | | |

| 00 | rd | op2 | imm22 |
|---|---|---|---|
| 00 | 00001 | 100 | 11 0000 0100 1111 0001 0101 |
| | %r1 | | 3 0 4 F 1 5 |

# Bitwise AND, set the Condition Codes (N, Z)

**andcc %r1, %r2, %r3**

**%r1 AND %r2 → %r3**

**sets the N and Z condition codes according to the result**

| 31 30 | 29–25 | 24–19 | 18–14 | 13 | 12–05 | 04–00 |
|---|---|---|---|---|---|---|
| 1 0 | rd | op3 | rs1 | 0 | 0 0 0 0 0 0 0 0 | rs2 |

| op3 (op=10) | |
|---|---|
| 010000 | addcc |
| 010001 | andcc |
| 010010 | orcc |
| 010110 | orncc |
| 100110 | srl |
| 111000 | jmpl |

| 10 | rd | op3 | rs1 | 0 | 0000 0000 | rs2 |
|---|---|---|---|---|---|---|
| 10 | 00011 | 010001 | 00001 | 0 | 0000 0000 | 00010 |
| | %r3 | | %r1 | | | %r2 |

# Bitwise OR, set the flags (N, Z)

orcc %r1, 1, %r1 ! Set the LSB in %r1 to 1

**OR the source operands put the result into the destination**

*%r1 OR 1 → %r1*

**sets the N and Z condition codes according to the result**

| 1 0 | rd | op3 | rs1 | 1 | simm13 |
|---|---|---|---|---|---|

| op3 (op=10) | |
|---|---|
| 010000 | addcc |
| 010001 | andcc |
| 010010 | orcc |
| 010110 | orncc |
| 100110 | srl |
| 111000 | jmpl |

| 10 | rd | op3 | rs1 | 1 | simm13 |
|---|---|---|---|---|---|
| 10 | 00001 | 010010 | 00001 | 1 | 0 0000 0000 0001 |
| | %r1 | | %r1 | | 1 |

# orcc %r1, 1, %r1 – bit to bit ORing

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| %r1 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| OR 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

bit to bit OR

%r1 AND 1 (bit to bit ORing)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| %r1 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1 |

result

# Bitwise NOR, set the flags (N, Z)

**orncc %r1, %r0, %r1** ! Complement r1

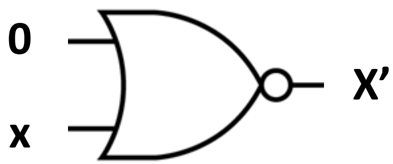
**OR the source operands put the result into the destination**

***%r1 ORNCC %r0 → %r1***

**set the N and Z condition codes according to the result**

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

| 31 | 30 | 29–25 | 24–19 | 18–14 | 13 | 12–05 | 04–00 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | rd | op3 | rs1 | 0 | 0 0 0 0 0 0 0 0 | rs2 |




| op3 (op=10) | |
|---|---|
| 010000 | addcc |
| 010001 | andcc |
| 010010 | orcc |
| 010110 | orncc |
| 100110 | srl |
| 111000 | jmpl |

| 10 | rd | op3 | rs1 | 0 | 0000 0000 | rs2 |
|---|---|---|---|---|---|---|
| 10 | 00001 | 010110 | 00001 | 0 | 0000 0000 | 00000 |
| | %r1 | | %r1 | | | %r0 |

# orncc %r1, %r0, %r1 – bit to bit NORing

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| %r1 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | X | X | X | X | |
| NOR | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | bit to bit NOR |
| %r0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

%r1 NOR %r0 (bit to bit NORing)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| %r1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | result |

$\underbrace{\hspace{10em}}_{\text{Complement of \%r1}}$

**Complement of %r1**

# SHIFT RIGHT LOGIC - (srl)

Shift a register to right by 0 to 31 bits

0s inserted from left

| | | |
|---|---|---|
| srl | %r1, 3, %r2 | !shift %r1 to right by 3 bits result to %r2 |
| srl | %r1, %r4, %r5 | !shift %r1 right by the value stored in %r4<br>!and store in %r5 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | rd | op3 | rs1 | 1 | simm13 |

| op3 (op=10) | |
|---|---|
| 010000 | addcc |
| 010001 | andcc |
| 010010 | orcc |
| 010110 | orncc |
| 100110 | srl |
| 111000 | jmpl |

| 10 | rd | op3 | rs1 | 1 | simm13 |
|---|---|---|---|---|---|
| 10 | 00010 | 100110 | 00001 | 1 | 0 0000 0000 0011 |
| | %r2 | | %r1 | | 3 |

# addcc

**addcc**: - adds the source operands into the destination operand using two's complement rep

- sets the condition codes according to the result

Example usage:

```
addcc %r1, %r2, %r4        !%r4 = %r1 + %r2
addcc %r1, 2, %r2          !%r2 = %r1 + 2
```

# addcc

**addcc %r1, 5, %r1**

| 1 0 | rd | op3 | rs1 | 1 | simm13 |
|---|---|---|---|---|---|

| op3 (op=10) | |
|---|---|
| 010000 | addcc |
| 010001 | andcc |
| 010010 | orcc |
| 010110 | orncc |
| 100110 | srl |
| 111000 | jmpl |

| 10 | rd | op3 | rs1 | 1 | simm13 |
|---|---|---|---|---|---|
| 10 | 00001 | 010000 | 00001 | 1 | 0 0000 0000 0101 |
| | %r1 | | %r1 | | 5 |

# CALL

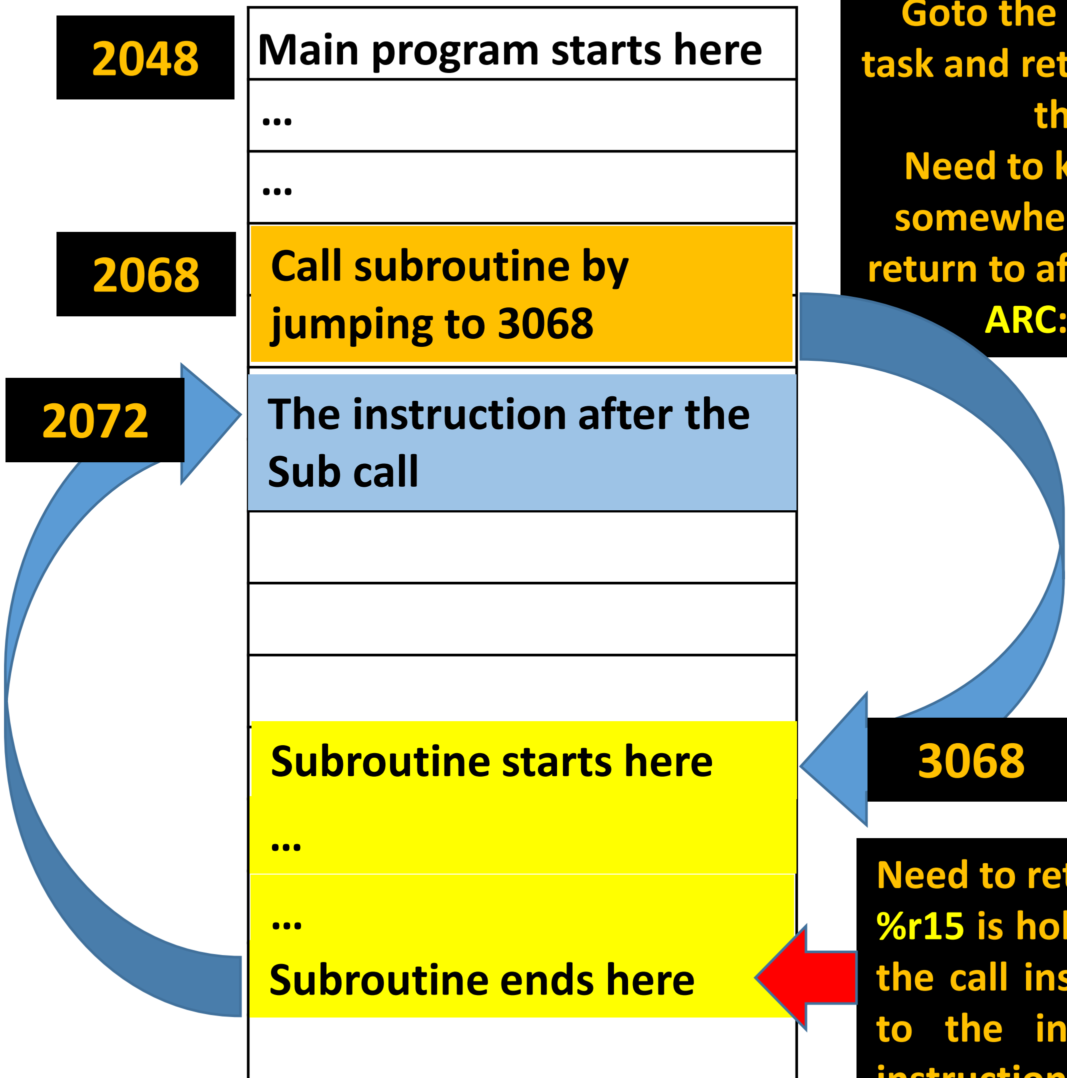
1. Call a subroutine
2. Store the address of the current instruction in **%r15**

The address of the next instruction to be executed is calculated by adding **4 x DISP30** to the address of the current instruction

# Calling A Subroutine

- When calling a subroutine we need to keep the address of the current location somewhere to be used after the subroutine is over
- In ARC, we keep this address in: **%r15**
- After the subroutine is over, **%r15 + 4**
- Indicates the address of the next instruction to be executed

# Calling A Subroutine & jmpl %r15+4




Goto the subroutine perform the task and return to continue executing the main program
Need to keep this address (2068) somewhere so we know where to return to after the subroutine is over
ARC: keep 2068 in %r15

Typically we don't use the address of the subroutine we use the Distance (displacement) that is 1000 bytes.
In ARC, 250 words
Goto 2068 + 250 x 4

Need to return to the main program. %r15 is holding 2068 (the address of the call instruction). We need to go to the instruction after the call instruction that is in address %r15+4 (2068 + 4 = 2072)

# call

**call sub_r** ! Calling subroutine, 100 bytes – 25 words – farther

| 31 | 30 | 29 … 00 |
|---|---|---|
| 0 | 1 | disp30 |

| | |
|---|---|
| 01 | Disp30 |
| 01 | 00 0000 0000 0000 0000 0000 0001 1001 |

25

***Note:***

***the operand is not the address of the subroutine***
***It is the distance (displacement) between the present instruction and the first instruction of the subroutine divided by 4***

# JUMP and LINK - jmpl

- Return from the subroutine
- Unconditional jump
- Jump to a new address
- Discard the current address

| op3 (op=10) | |
|---|---|
| 010000 | addcc |
| 010001 | andcc |
| 010010 | orcc |
| 010110 | orncc |
| 100110 | srl |
| 111000 | jmpl |

**Jmpl %r15 + 4, %r0**

| | | | | | |
|---|---|---|---|---|---|
| 1 0 | rd | op3 | rs1 | 1 | simm13 |

| 10 | rd | op3 | rs1 | 1 | simm13 |
|---|---|---|---|---|---|
| 10 | 00000 | 111000 | 01111 | 1 | 0 0000 0000 0100 |
| | %r0 | | %r15 | | 4 |

# BRANCH IF EQUAL (be)

Branch of equal to ZERO

- If Z flag is set
- Go to the address ( **4 x DISP22** **+ current address**)

- If Z flag is not set
- Go to the next instruction

# Branch if equal to 0 (be)

**be** **label** !label is 20 bytes (**5** words) farther

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | cond | op2 | disp22 |

| cond | branch |
|---|---|
| 0001 | be |
| 0101 | bcs |
| 0110 | bneg |
| 0111 | bvs |
| 1000 | ba |

| op2 | Inst. |
|---|---|
| 010 | branch |
| 100 | sethi |

| 00 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|
| 00 | 0 | 0001 | 010 | 00 0000 0000 0000 0000 0101 |

5

# BRANCH IF NEGATIVE (beng)

Jump to a new address

If the condition **n** (negative) is set (**n = 1**)

How to calculate the address?

Address = **4 x DISP22** + Address of the current instruction

If **n = 0**, then go to the instruction that follows beng

# Branch if negative (bneg)

**bneg** **label** !jump to **5** words farther

| 0 | 0 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|---|

| cond | branch |
|---|---|
| 0001 | be |
| 0101 | bcs |
| 0110 | bneg |
| 0111 | bvs |
| 1000 | ba |

| op2 | Inst. |
|---|---|
| 010 | branch |
| 100 | sethi |

| 00 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|
| 00 | 0 | 0110 | 010 | 00 0000 0000 0000 0000 0101 |

5

# BRANCH IF CONDITION CODE (bcs)

Branch if condition code is set

Then jump to the new address

Otherwise go to the next instruction after **bcs**

New address = **4 x DISP22** + address of the current instruction

# bcs

**bcs label** !jump to 5 words farther

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | cond | op2 | disp22 |

| cond | branch |
|---|---|
| 0001 | be |
| 0101 | bcs |
| 0110 | bneg |
| 0111 | bvs |
| 1000 | ba |

| op2 | Inst. |
|---|---|
| 010 | branch |
| 100 | sethi |

| 00 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|
| 00 | 0 | 0101 | 010 | 00 0000 0000 0000 0000 0101 |

5

# BRANCH IF OVERFLOW FLAG (bvs)

Branch if the overflow flag is set

Then jump to the new address

Otherwise go to the next instruction after **bvs**

New address = **4 x DISP22** + address of the current instruction

# bvs

**bvs label** !jump to 5 words farther

| 0 | 0 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|---|

| cond | branch |
|---|---|
| 0001 | be |
| 0101 | bcs |
| 0110 | bneg |
| 0111 | bvs |
| 1000 | ba |

| op2 | Inst. |
|---|---|
| 010 | branch |
| 100 | sethi |

| 00 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|
| 00 | 0 | 0111 | 010 | 00 0000 0000 0000 0000 0101 |

5

# BRANCH (ba)

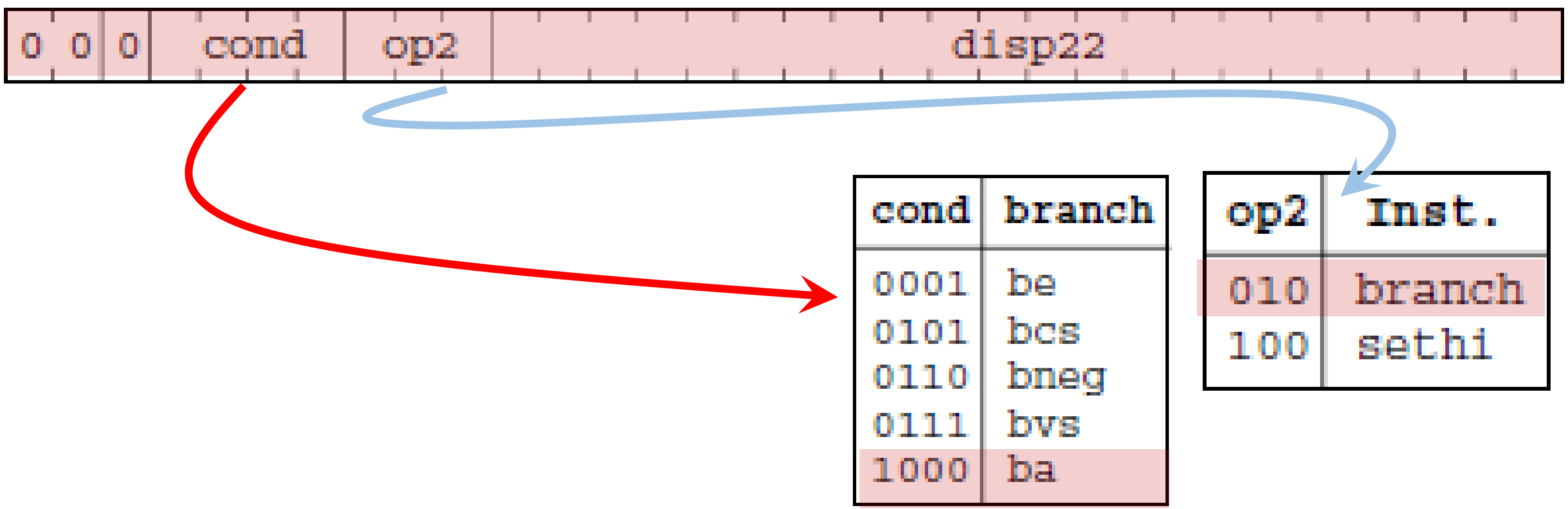
Branch (no condition)

Branch address = **4 x DISP22** +

address of the current instruction

# ba

**ba label** !jump to 5 words farther




| 00 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|
| 00 | 0 | 1000 | 010 | 00 0000 0000 0000 0000 0101 |

5

# ba

**ba label** !jump to 5 words earlier

!displacement (**-5**)

| 0 | 0 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|---|

| cond | branch |
|---|---|
| 0001 | be |
| 0101 | bcs |
| 0110 | bneg |
| 0111 | bvs |
| 1000 | ba |

| op2 | Inst. |
|---|---|
| 010 | branch |
| 100 | sethi |

-5 representation in 1's and 2's comp

| | | | | | | |
|---|---|---|---|---|---|---|
| 5: | 00 | 0000 | 0000 | 0000 | 0000 | 0101 |
| -5: | 11 | 1111 | 1111 | 1111 | 1111 | 1010 |
| -5: | 11 | 1111 | 1111 | 1111 | 1111 | 1011 |

| 00 | 0 | cond | op2 | disp22 |
|---|---|---|---|---|
| 00 | 0 | 1000 | 010 | 11 1111 1111 1111 1111 1011 |

- 5