MICROVA

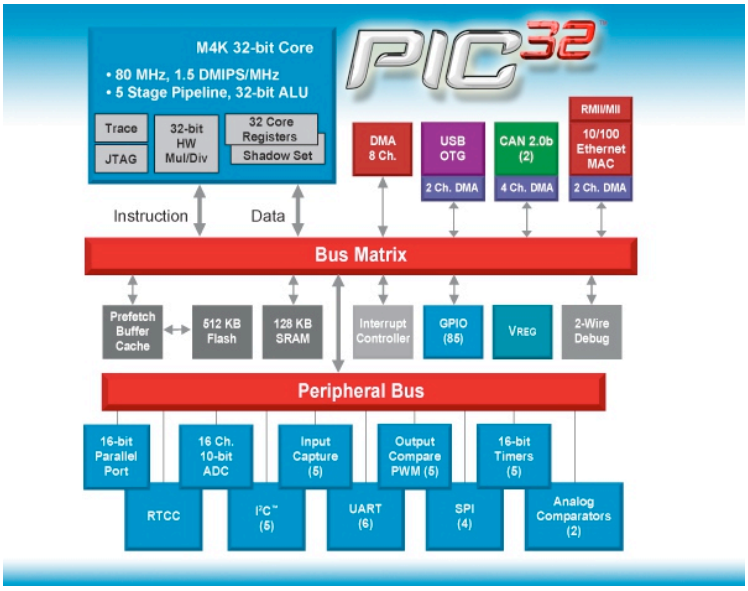
Microva Corporation

MIPS32

Assembler Manual 2.6

(For Microchip PIC32MX/MZ Microcontrollers)

MIPS TECHNOLOGIES



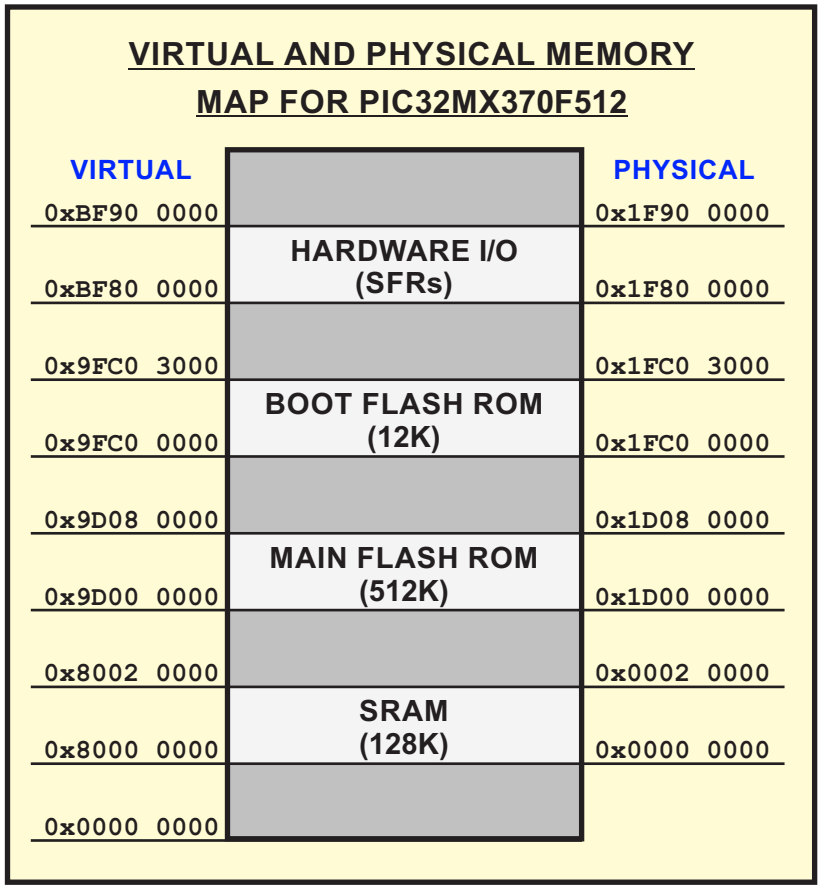
MICROCHIP

1.0 Introduction

This is the programmer's reference manual for the Microva assembly software which creates standard 32 bit op codes for MIPS32 type CPUs using simple "pseudo" op code mnemonics which are very similar to the original Motorola 68000 op codes. The Microva BASIC real time kernel firmware was written using this assembly software.

2.0 Memory Map and Operating Mode

MIPS CPUs include a Memory Management Unit (MMU) which creates virtual memory addressing in order to distinguish between “Supervisor” mode addressing (also called Kernel mode) and “User” mode addressing. Most hardware resources are not accessible while in User mode in order to prevent User programs from illegal activity (whether intentional or not). MIPS CPUs use one or more “coprocessors” to handle hardware configuration, floating point math, interrupts and exceptions (such as RESET). The CPU resets in Kernel mode. The base operating mode can be changed from Kernel to User mode by writing a bit in CP0 (coprocessor 0). All exceptions and interrupts force the CPU into Kernel mode and this is the only method to change from User to Kernel mode since User mode software cannot access CP0.



User mode program addresses correspond to the lower half of the 4GB memory space (i.e. when addr b31 = 0) and Kernel mode program addresses use the upper 2GB of the 4GB memory address range. The User mode address space is also the same as the physical memory address space (except of course User mode programs cannot read/write to the hardware I/O memory locations). The virtual and physical memory addresses are shown in the graphic for the PIC32MX370F512 CPU. The Microva BASIC kernel firmware always runs in Kernel mode since interrupt software forces the CPU to run in Kernel mode and since User mode software cannot access the hardware I/O. This means that the assembler source program must use the virtual memory address in the “org” statement so that the assembler can calculate the correct address to read, for example, data from a table in ROM. However, the Intel HEX output file must use the physical memory addresses so that the assembled op codes can be properly loaded into the chip programmer. Therefore, the assembler software always clears b31 in the “org” statement before writing the HEX output file. Also, any ROM files (such as Fonts or bitmaps) must be loaded into memory at the physical memory address not the virtual address (but are then referenced in the source program at the virtual address).

3.0 Registers, The Stack and Subroutines

The stack is "full descending" which means new data pushed onto the stack resides at progressively lower addresses. There are (32) registers available in the MIPS32 CPU. All registers are (32) bit and all registers may be used with all op codes. The Program Counter (PC) is not one of these registers. The following registers are reserved for special functions:

r0 = always 0 (by hardware)
r29 = used by the assembler to hold intermediate results
r30 = stack pointer
r31 = link register

The link register (r31) is used to hold the return address during a subroutine call. There are two types of subroutines- those that call other subroutines (i.e. “nested” subroutines) and “stand alone” subroutines. Each nested subroutine must "push r31" (and optionally any other registers) as the first instruction in the subroutine. The return instruction from a nested subroutine (”retn”) will pop r31 from the stack and copy it to the PC. Stand alone subroutines do not need to “push r31” and the return instruction for a stand alone subroutine should be “goto r31” (stand alone subroutines can still use the “push r31/retn” statements but it takes less CPU clock cycles to use the simpler “goto r31”). Subroutines can be nested as often as needed (until the stack overflows). See the examples below.

EXAMPLE STAND ALONE SUBROUTINE

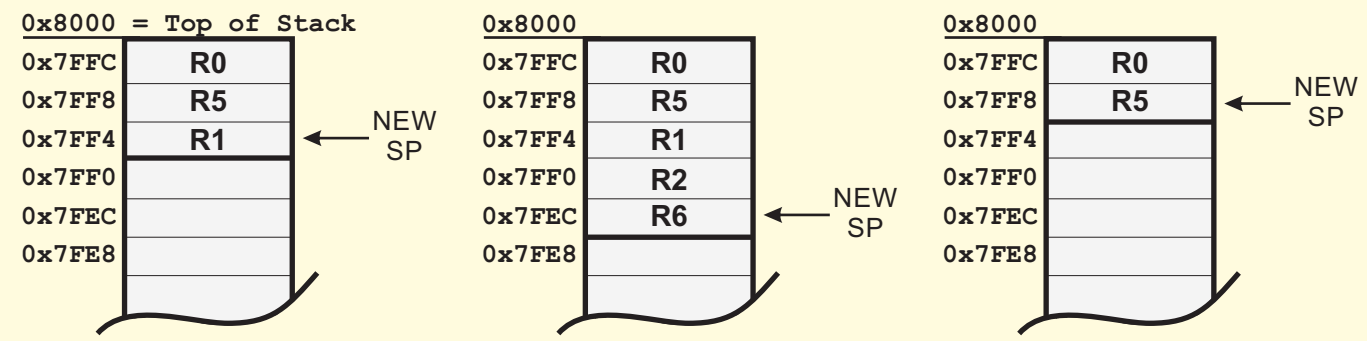
```
SetBit3:
  bset  3, r5
  goto  r31
```

EXAMPLE NESTED SUBROUTINE

```
SetBit3_5:
  push  r31
  call  SetBit3
  bset  5, r5
  retn
```

STACK OPERATION

Assumes Top of RAM = 0x7FFF



4.0 Op Code and Data Format

The Microchip PIC32MX MCUs use the "little endian" memory format. This means that data is stored with the LSB at the lowest address. The example below illustrates the little endian data format when storing two data words. The op codes shown on the following pages are listed alphabetically by the pseudo op code. The actual MIPS32 op code is described and shown below the pseudo op code. The “number of clocks” shown for each pseudo op code is the number of CPU clock cycles needed to execute the op code when running out of zero wait state program memory. Just as in the Motorola 68000, the primary op code to move data around is the MOV command. Data can be moved a byte at a time using MOVB or a half word at a time (16 bits) using MOVH or a full word at a time (32 bits) using MOVL. The move address must be data aligned, i.e. even for MOVH and modulo 4 for MOVL.

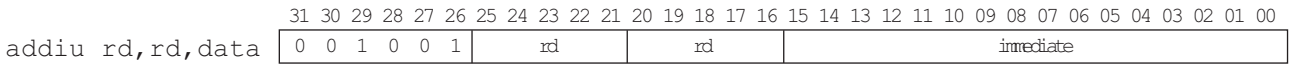
LITTLE ENDIAN DATA FORMAT

```
org 0x7C00
dl  0x826B0A51
dl  0x27054A3C
```

ADDR	MEMORY
0x7C07	0x27
0x7C06	0x05
0x7C05	0x4A
0x7C04	0x3C
0x7C03	0x82
0x7C02	0x6B
0x7C01	0x0A
0x7C00	0x51

add data, rd

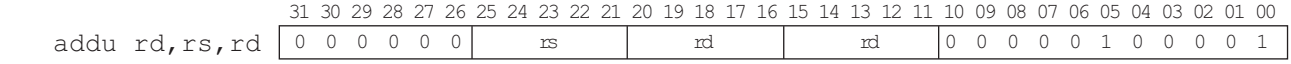
rd = rd + data (data must range from -32,768 to +32,767)
number clocks: 1



---

add rs, rd

rd = rd + rs
number clocks: 1



---

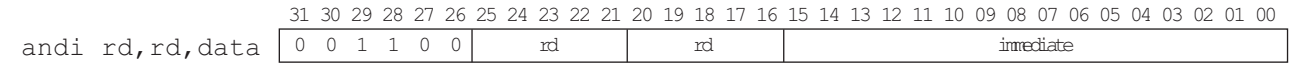
align

assembler directive which moves the instruction address pointer to the next "modulo 4" address by inserting 2, 3, or 4 data bytes (all = 0x00) or by doing nothing if the address is already "modulo 4"

---

and data, rd

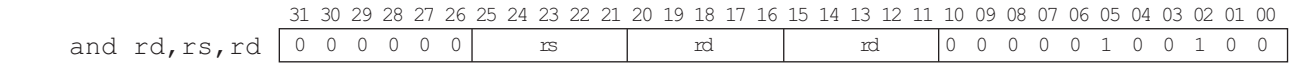
rd = rd and data (data must range from 0 to 0xFFFF)
number clocks: 1



---

and rs, rd

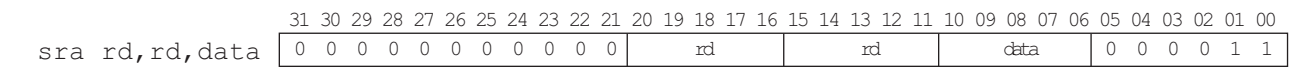
rd = rd and rs
number clocks: 1



---

asr n, rd

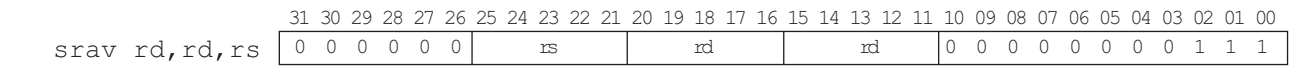
rd = rd asr n times (n must range from 0 to 31)
number clocks: 1



---

asr rs, rd

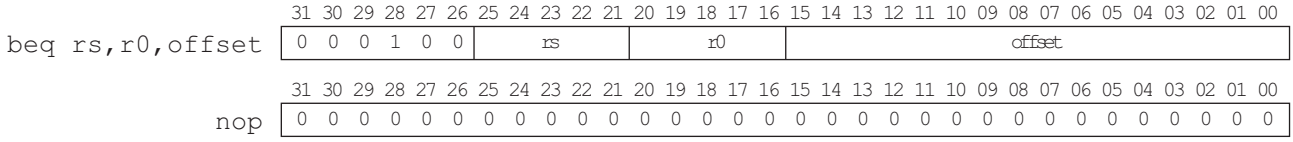
rd = rd asr "rs" times (lower 5 bits in "rs" are used for the shift data)
number clocks: 1



---

beq rs, label

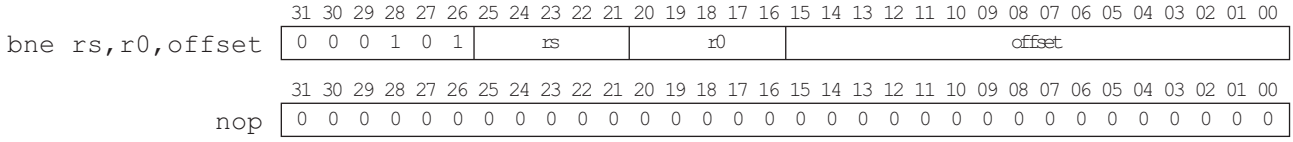
branch if rs equals zero, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

bne rs, label

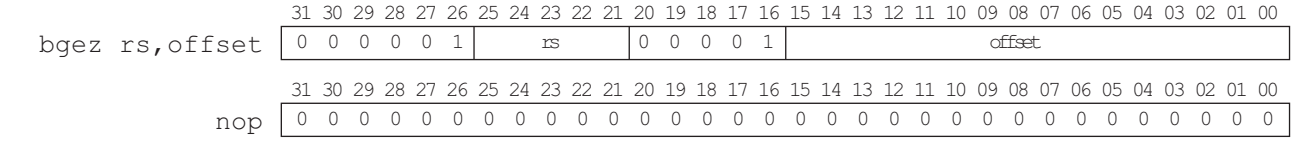
branch if rs not equal zero, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

bge rs, label

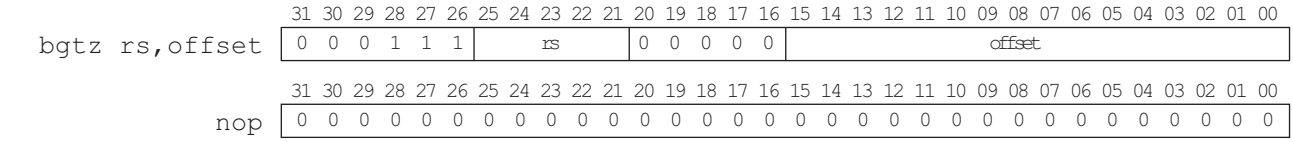
branch if rs is greater than or equal zero, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

bgt rs, label

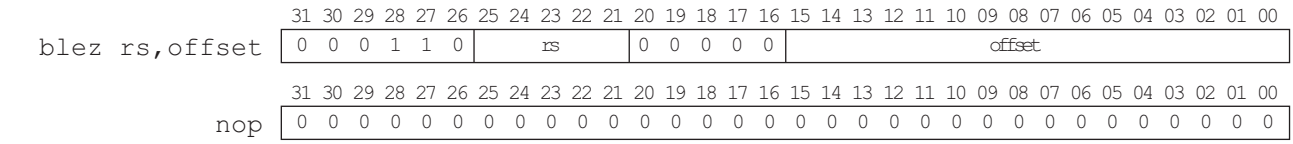
branch if rs is greater than zero, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

ble rs, label

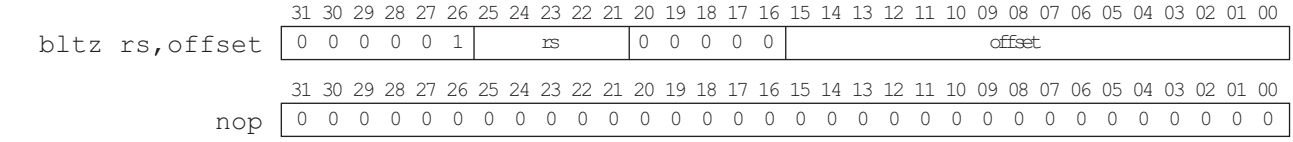
branch if rs is less than or equal zero, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

blt rs, label

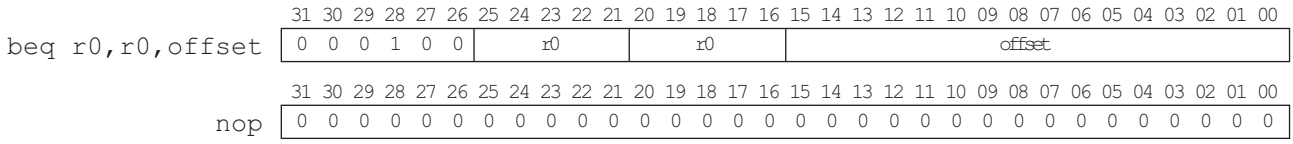
branch if rs is less than zero, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

bra label

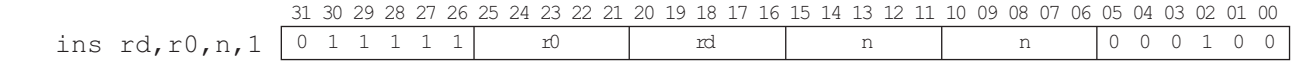
unconditional branch, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

bclr n, rd

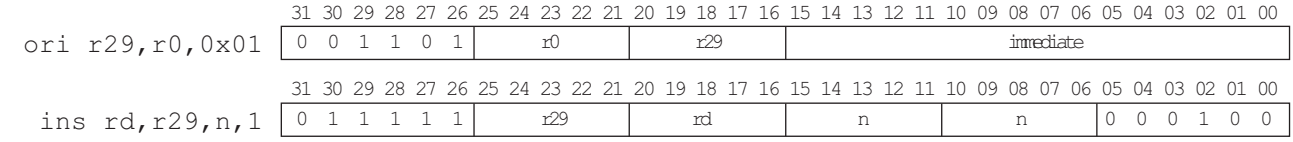
clear bit n of rd (n must range from 0 to 31)
number clocks: 1



---

bset n, rd

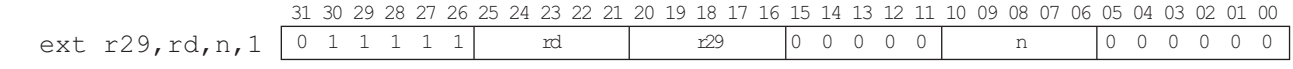
set bit n of rd (n must range from 0 to 31)
number clocks: 2



---

btst n, rd

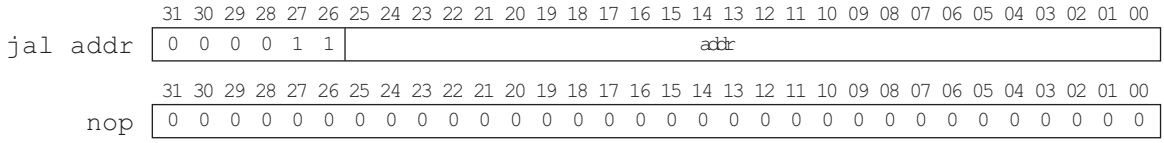
load bit n of rd to r29 (n must range from 0 to 31)
number clocks: 1



---

call label

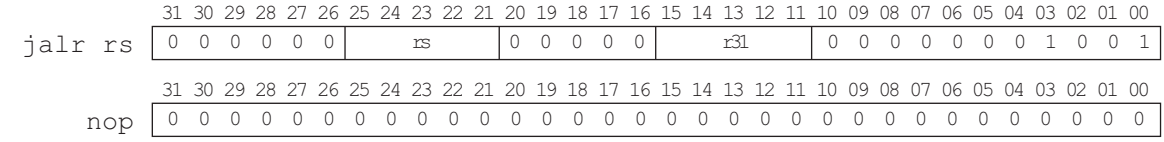
call a subroutine (addr = (label addr & 0x0FFF FFFF) / 4)
number clocks: 2



---

call rs

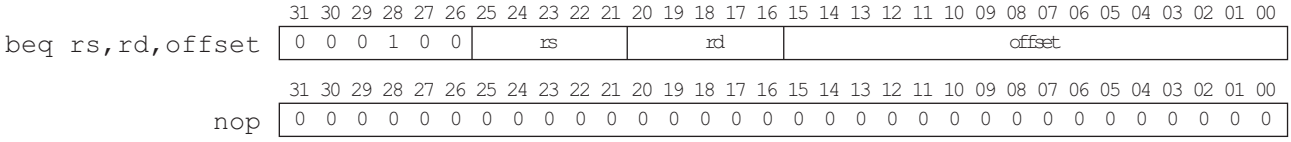
call a subroutine using addr in rs (addr must be modulo 4)
number clocks: 2



---

cbeq rs, rd, label

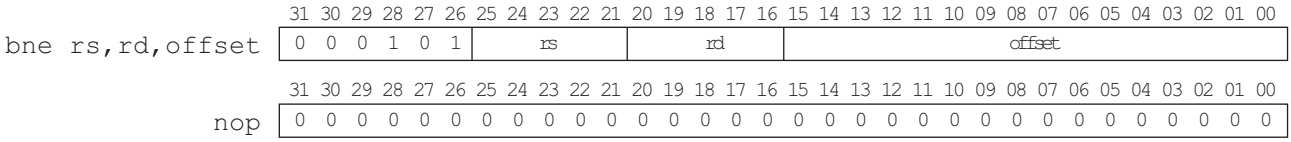
compare and branch if rs equals rd, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

cbne rs, rd, label

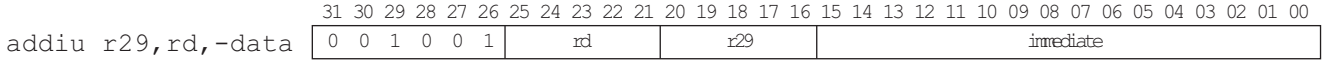
compare and branch if rs not equal rd, offset = (label addr - instruction addr - 4) / 4
number clocks: 2



---

cmp data, rd

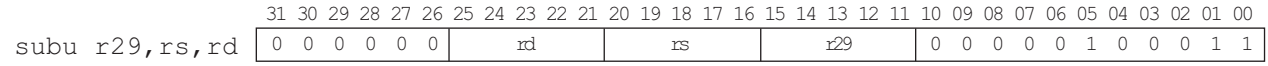
r29 = rd - data (data must range from -32,768 to +32,767)
number clocks: 1



---

cmp rs, rd

r29 = rd - rs
number clocks: 1



---

db <data>

assembler directive which inserts byte data in the program

```
db  0xFF, 5, "Hello", 23          ;ASCII strings are allowed
db  5, MyLabel                    ;Error:  labels not permitted
db  5, MyData                     ;equates are allowed
db  5, -23                        ;Error:  byte range = 0 to 255
```

---

dh <data>

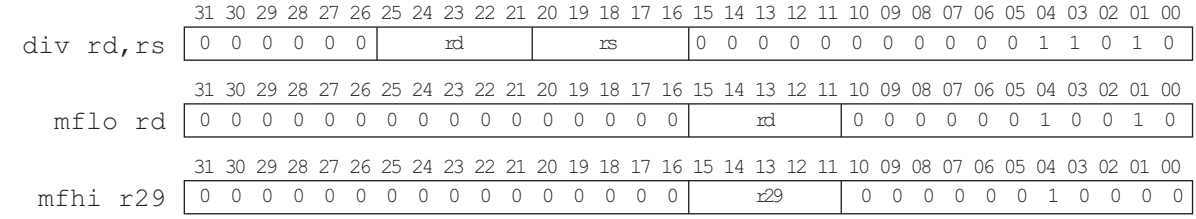
assembler directive which inserts "half integer" data in the program (data is stored in "little endian" format)

```
dh  5, MyLabel                    ;Error:  labels not permitted
dh  5, MyData                     ;equates are allowed
dh  5, -23                        ;Error:  range = 0 to 0xFFFF
```

---

divs rs, rd

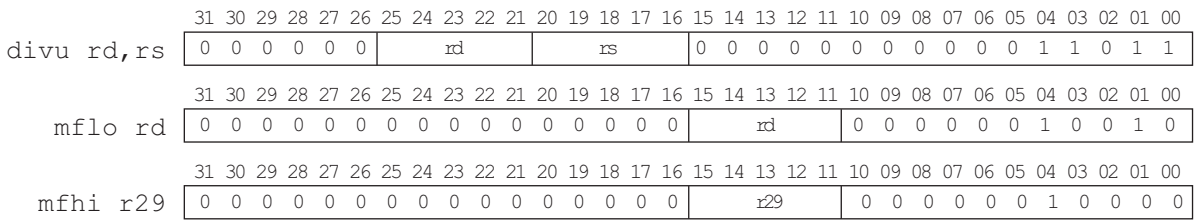
rd = rd / rs (both dividend rd and divisor rs are 32 bit signed integers, result rd is not rounded)
r29 = remainder
number clocks: 4-36



---

divu rs, rd

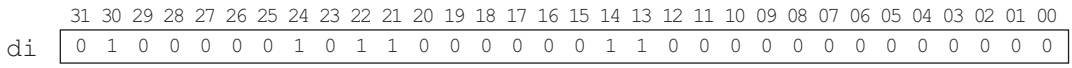
rd = rd / rs (both dividend rd and divisor rs are 32 bit unsigned integers, result rd is not rounded)
r29 = remainder
number clocks: 4-36



---

di

disable interrupts
number clocks: n



---

dl <data>

assembler directive which inserts long data in the program (data will be stored in "little endian" format)

```
dl  0xFF, "Hello"                ;Error: ASCII strings not allowed
dl  0xFF80FA55, 0xFFE0, MyLabel  ;labels and equates are allowed
```

---

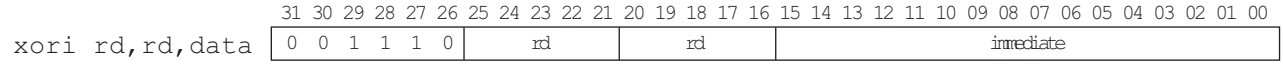
end

assembler directive which ends the program

---

eor data, rd

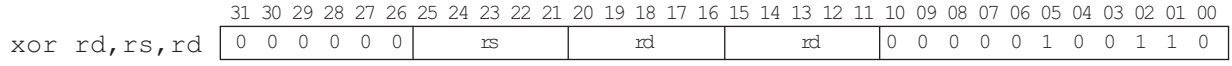
rd = rd eor data (data must range from 0 to 0xFFFF)
number clocks: 1



---

eor rs, rd

rd = rd eor rs
number clocks: 1



---

equ <name>, <data>

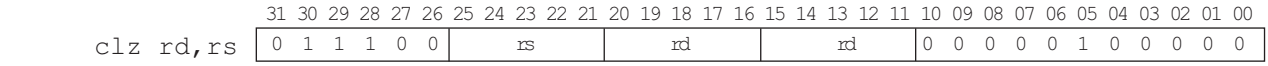
assembler directive used to assign equates

```
equ  MyData1, 0xFF                    ;MyData1 = 255
equ  MyData2, -15                     ;MyData2 = -15
equ  MyData3, MyAddr                  ;Error: labels not permitted
```

---

glz rs, rd

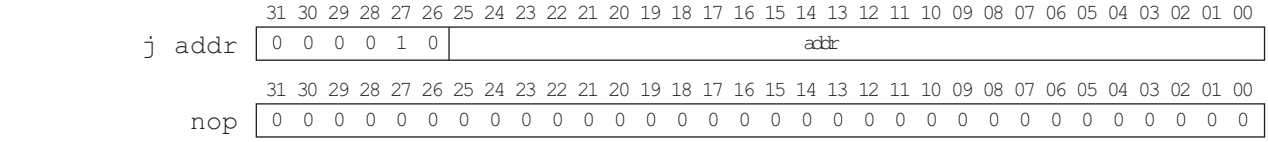
rd = number of “leading zeros” in rs (rd will range from 0 to 32)
number clocks: 1



---

goto label

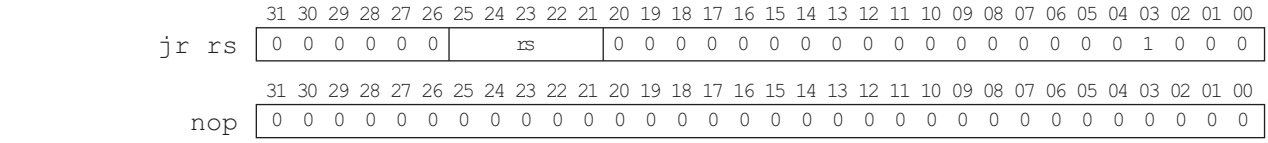
goto label (addr = (label addr & 0x0FFF FFFF) / 4)
number clocks: 2



---

goto rs

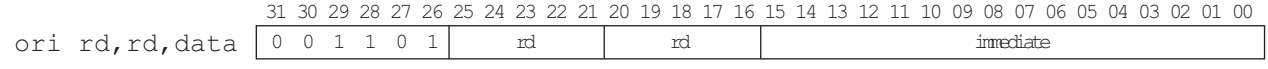
goto addr in rs (addr must be modulo 4)
number clocks: 2



---

ior data, rd

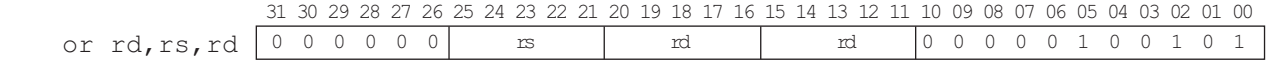
rd = rd or data (data must range from 0 to 0xFFFF)
number clocks: 1



---

ior rs, rd

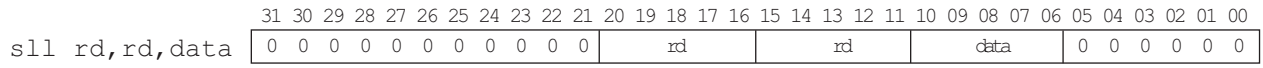
rd = rd or rs
number clocks: 1



---

lsl n, rd

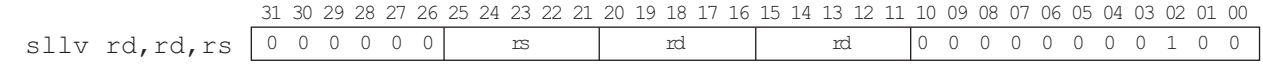
rd = rd lsl n times (n must range from 0 to 31)
number clocks: 1



---

lsl rs, rd

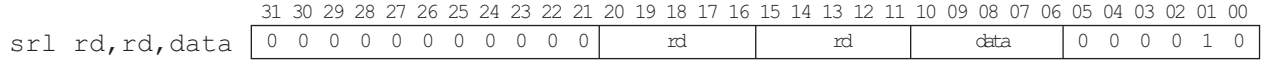
rd = rd lsl "rs" times (lower 5 bits in "rs" are used for the shift data)
number clocks: 1



---

lsr n, rd

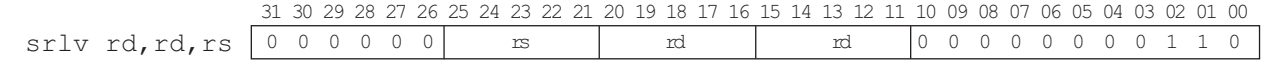
rd = rd lsr n times (n must range from 0 to 31)
number clocks: 1



---

lsr rs, rd

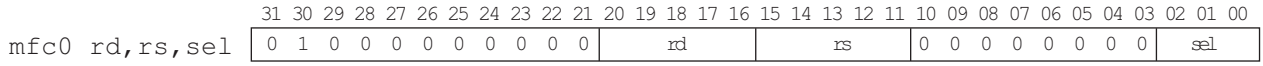
rd = rd lsr "rs" times (lower 5 bits in "rs" are used for the shift data)
number clocks: 1



---

mfc0 <cp0 reg>, rd

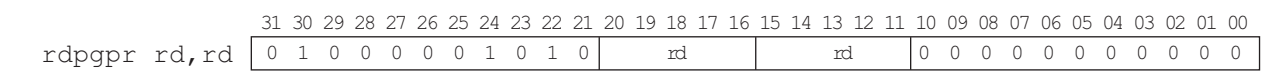
rd = CP0 register (reg name must be: Status, Count, Compare, IntCtl, SrsCtl, Cause, EPC, or EBase)
number clocks: n



---

mfpr rd

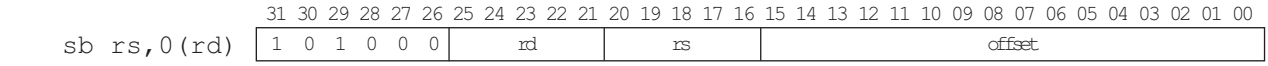
rd = general register from previous shadow register set
number clocks: n



---

movb rs, (rd)

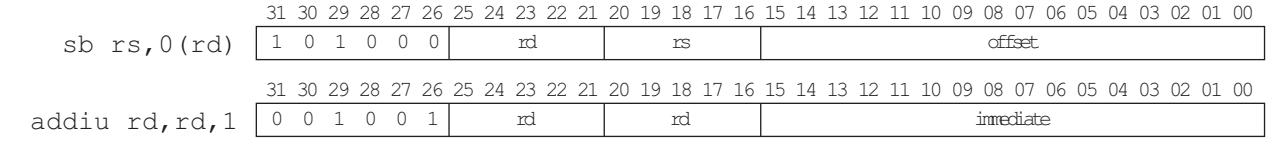
number clocks: 1



---

movb rs, (rd)+

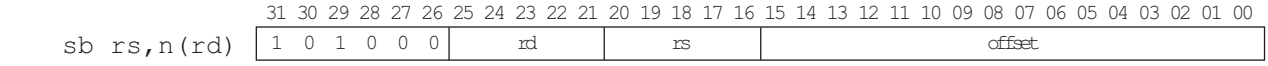
number clocks: 2



---

movb rs, n(rd)

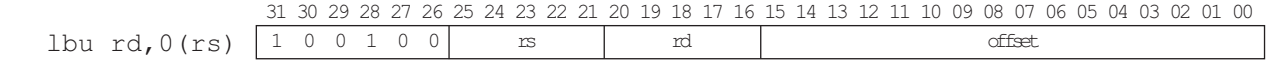
n is offset in bytes (offset must range from -32,768 to +32,767)
number clocks: 1



---

movb (rs), rd

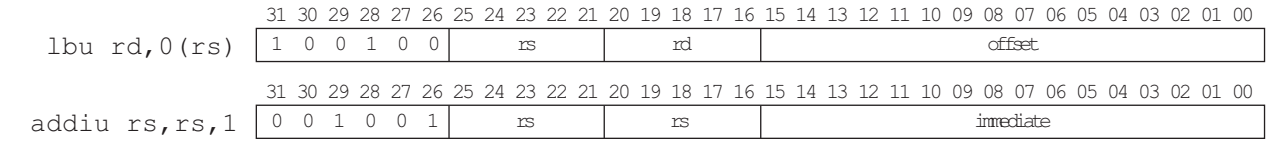
number clocks: 1



---

movb (rs)+, rd

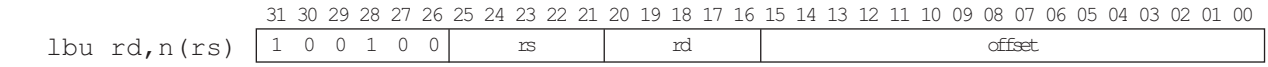
number clocks: 2



---

movb n(rs), rd

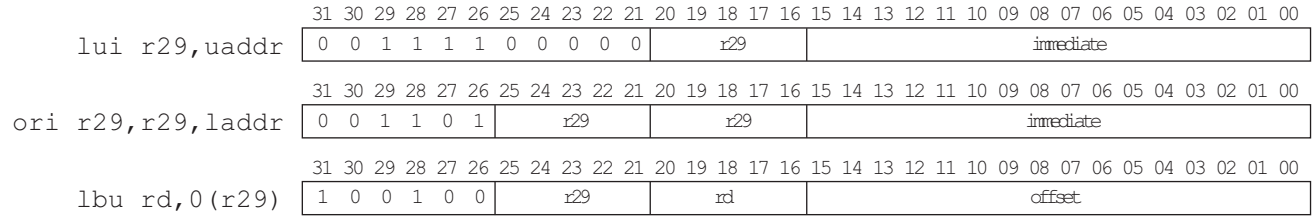
n is offset in bytes (offset must range from -32,768 to +32,767)
number clocks: 1



---

movb (addr), rd

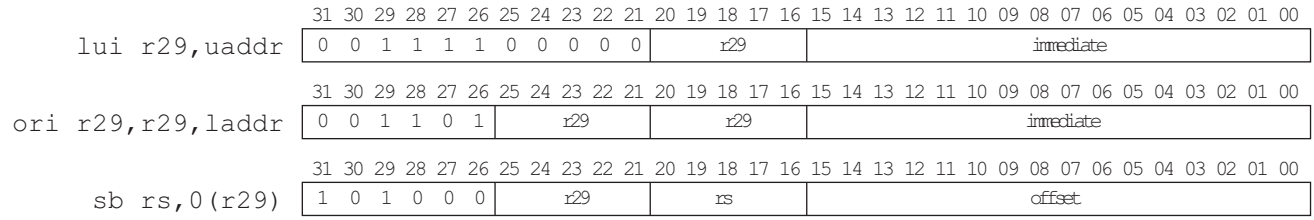
"addr" must be a constant
number clocks: 3



---

movb rs, (addr)

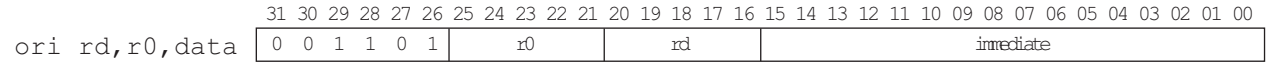
"addr" must be a constant
number clocks: 3



---

movh data, rd

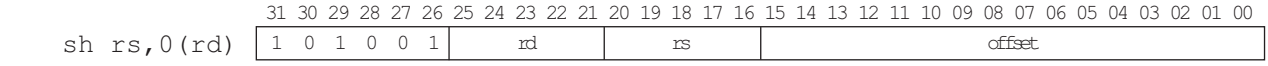
data must range from 0 to 0xFFFF
number clocks: 1



---

movh rs, (rd)

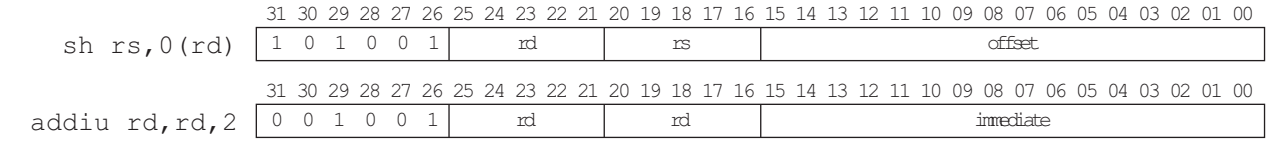
number clocks: 1



---

movh rs, (rd)+

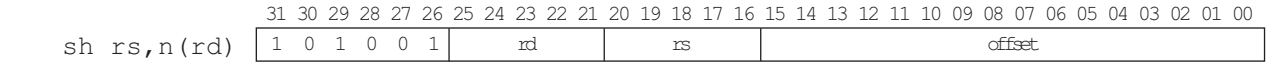
number clocks: 2



---

movh rs, n(rd)

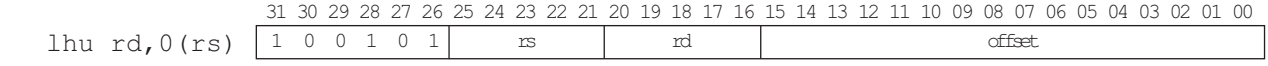
n is offset in half words (offset = n * 2 and must range from -32,768 to +32,767)
number clocks: 1



---

movh (rs), rd

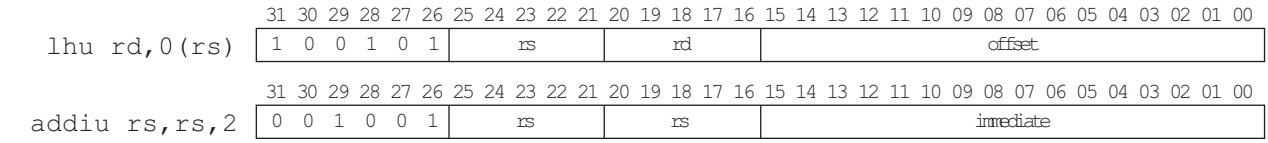
number clocks: 1



---

movh (rs)+, rd

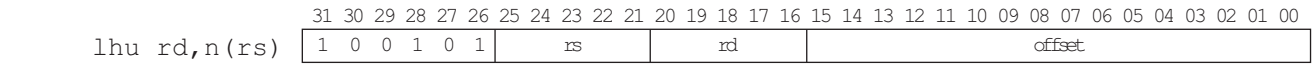
number clocks: 2



---

movh n(rs), rd

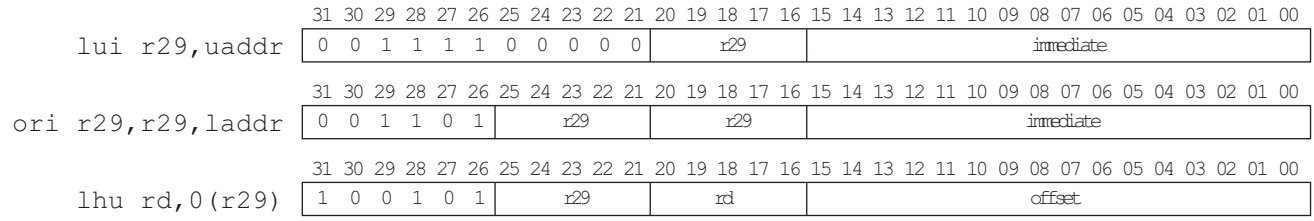
n is offset in half words (offset = n * 2 and must range from -32,768 to +32,767)
number clocks: 1



---

movh (addr), rd

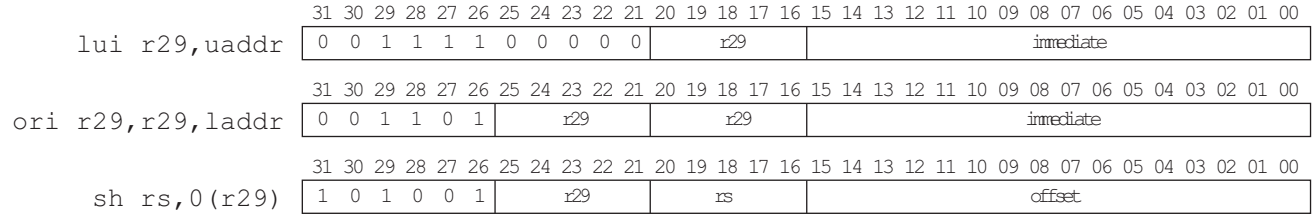
"addr" must be a constant
number clocks: 3



---

movh rs, (addr)

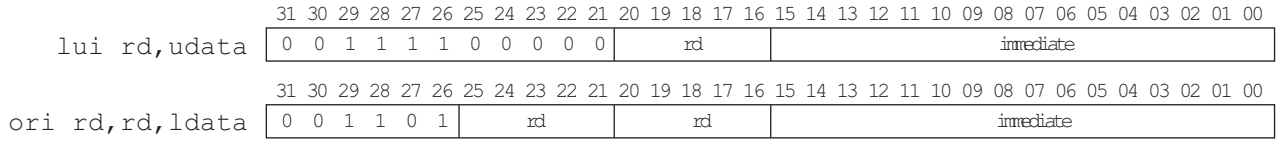
"addr" must be a constant
number clocks: 3



---

movl data, rd

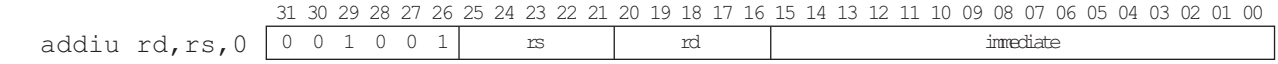
data can be a constant or an address label (range is -2,147,483,648 to +2,147,483,647)
number clocks: 2



---

movl rs, rd

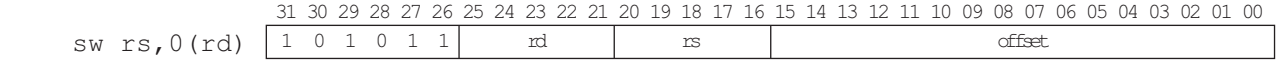
number clocks: 1



---

movl rs, (rd)

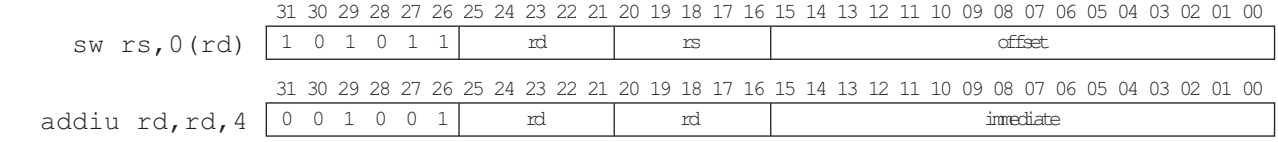
number clocks: 1



---

movl rs, (rd)+

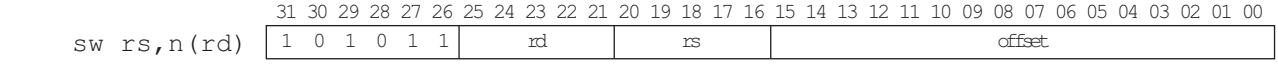
number clocks: 2



---

movl rs, n(rd)

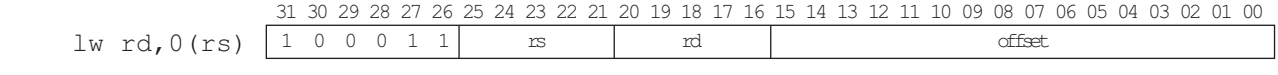
n is offset in words (offset = n * 4 and must range from -32,768 to +32,767)
number clocks: 1



---

movl (rs), rd

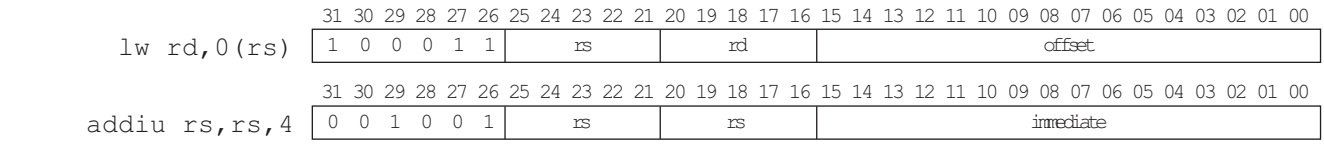
number clocks: 1



---

movl (rs)+, rd

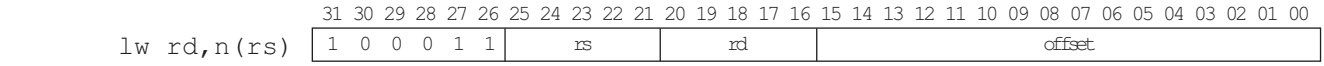
number clocks: 2



---

movl n(rs), rd

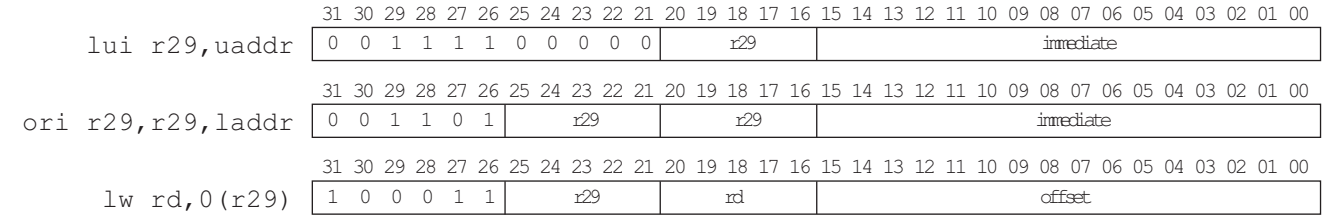
n is offset in words (offset = n * 4 and must range from -32,768 to +32,767)
number clocks: 1



---

movl (addr), rd

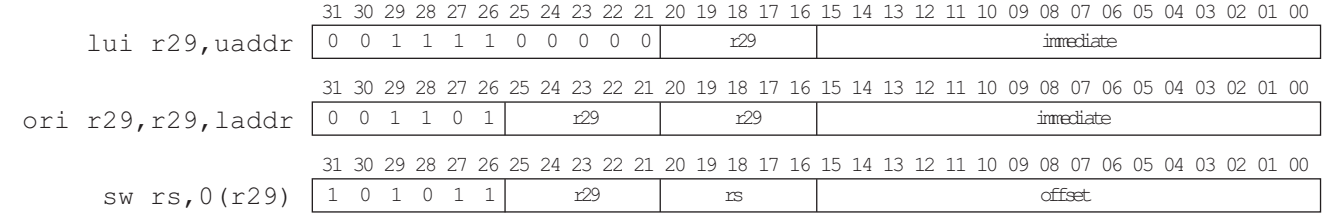
"addr" must be a constant
number clocks: 3



---

movl rs, (addr)

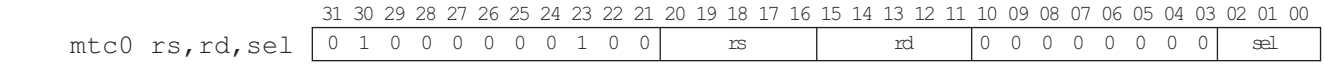
"addr" must be a constant
number clocks: 3



---

mtc0 rs, <cp0 reg>

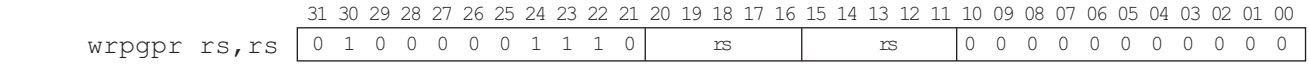
CP0 register = rs (CP0 reg names: Status, Config, Count, Compare, IntCtl, SrsCtl, Cause, EPC, or EBase)
number clocks: 1



---

mtpr rs

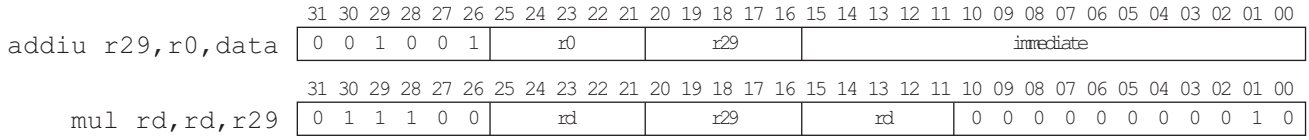
move general register rs to previous shadow register set
number clocks: n



---

mul data, rd

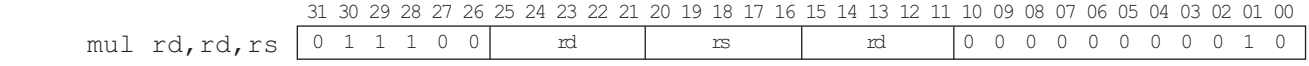
rd = data * rd (both data and rd can be positive or negative, data must range from -32,768 to +32,767)
number clocks: 3



---

mul rs, rd

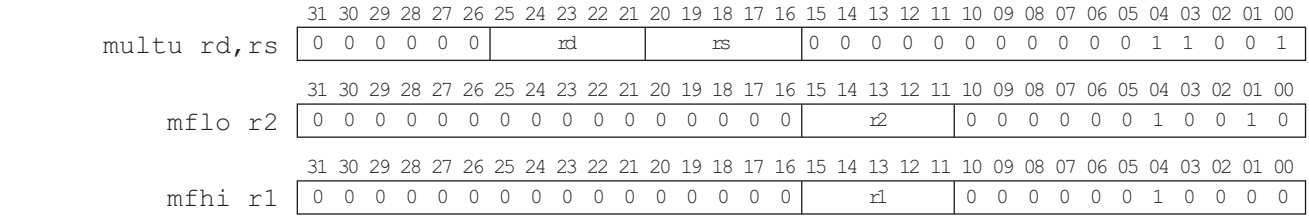
rd = rd * rs (both rd and rs can be positive or negative)
number clocks: 2



---

mulx rs, rd

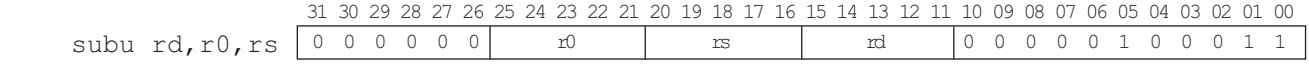
r1/r2 = rd * rs (both rd and rs are 32 bit unsigned integers, result = 64 bits, r1 = b63 to b32, r2 = b31 to b0)
number clocks: 4



---

neg rs, rd

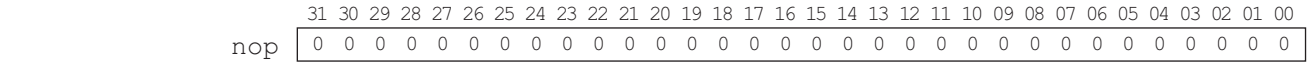
rd = 0 - rs
number clocks: 1



---

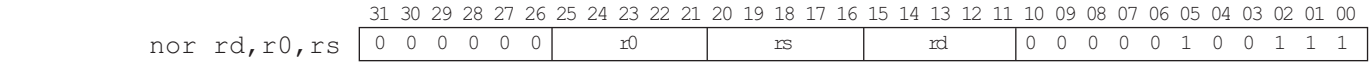
nop

no operation
number clocks: 1



not rs, rd

rd = rs eor 0xFFFFFFFF
number clocks: 1



---

org address

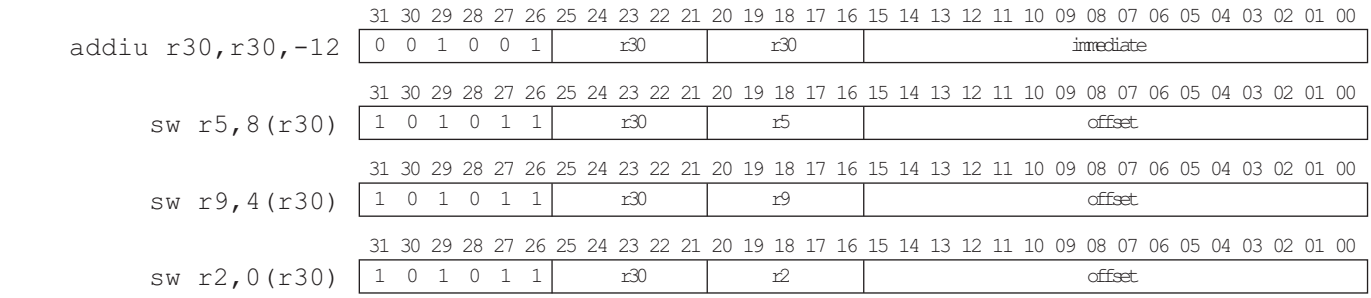
assembler directive assigns the starting address for the output file from the first "org" statement which must be the first non-comment line in the source program
address must be modulo 16 and address bit 31 will be cleared when writing the Intel hex output file in order to load the file into physical not virtual memory
additional "org" statements will append data = 0xFF to the output file in order to move the address pointer to the new "org" statement address location so that those unused locations remain erased in Flash ROM

---

push register list

"register list" consists of any combination of r0,r1,r2,...,r31 in the order they will be saved to the stack
number clocks: 1 + num regs

example OP CODE for: push r5,r9,r2

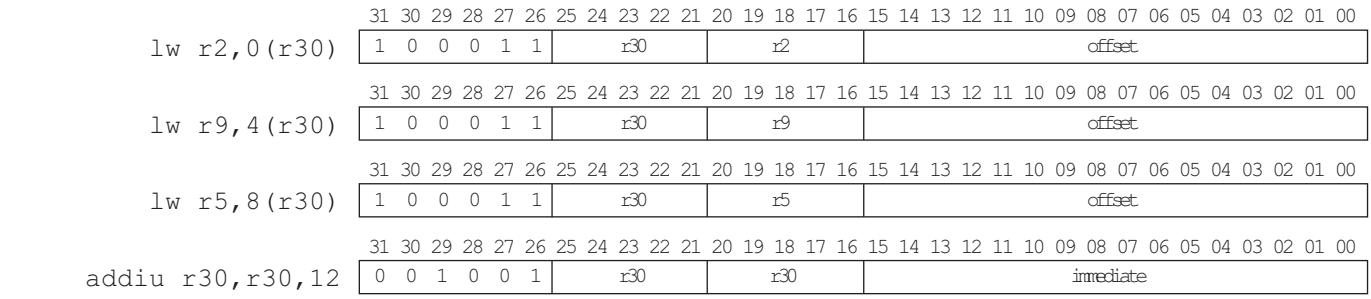


---

pop register list

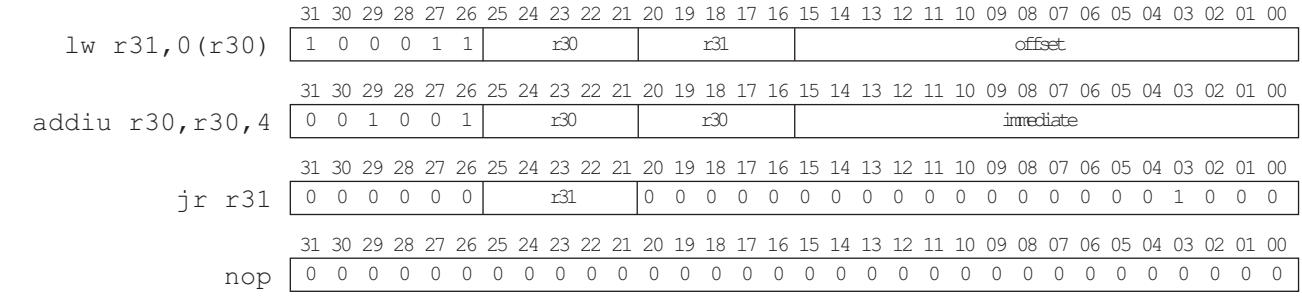
"register list" consists of any combination of r0,r1,r2,...,r31 in the order they will be returned from the stack
number clocks: 1 + num regs

example OP CODE for: pop r2,r9,r5



retn

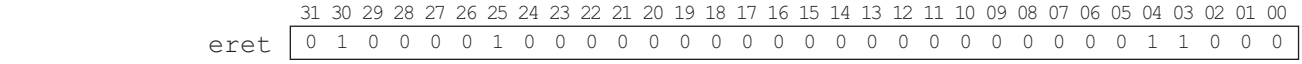
return from subroutine
number clocks: 5



---

reti

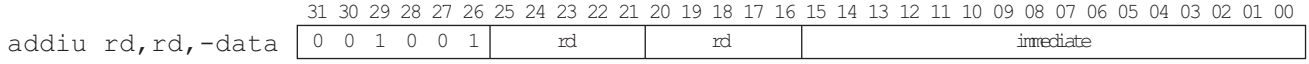
return from interrupt
number clocks: n



---

sub data, rd

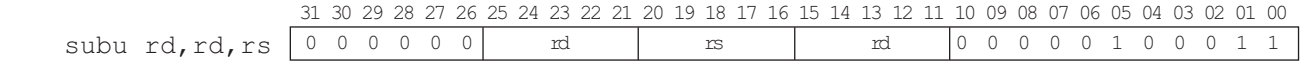
rd = rd - data (data must range from +32,768 to -32,767)
number clocks: 1



---

sub rs, rd

rd = rd - rs
number clocks: 1



---

wait

start low power mode (sleep or idle)
number clocks: 1