Assembly Language in NIOS II

Reminders

- Assembly demo due Friday
- No quiz this week
- No lab due this week
  - **Get your boards if you have already done so**
- Lab 1 due next week at the beginning of your lab session
  - **See lab for late penalties**
  - **Signoffs can be received in first 15 minutes of lab session or during office hours**

What is Assembly Language?

- Low-level programming language for a computer
- One-to-one correspondence with the machine instructions
- Specific to a given processor
- Uses:
  - **Mnemonics to represent the names of low-level machine instructions**
  - **Labels to represent the names of variables or memory addresses**
  - **Directives to define data and constants**
  - **Macros to facilitate the inline expansion of text into other code**

Directives

- Provide information to the assembler
  - **Do not correspond with an assembly language instruction or data**
  - **Always begin with .**
- Commonly used directives
  - **.ascii “*string*”**
    - A string of ASCII characters is loaded into consecutive byte addresses in the memory. Multiple strings, separated by commas can be specified
  - **.asciz *“string”***
    - Same as .ascii, except that each string is terminated by a zero byte
  - **.data**
    - Identifies the data that should be placed in the data section of the memory

Directives

- Commonly used directives
  - **.end**
    - Marks the end of the source code file
  - **.equ *symbol, expression***
    - Sets the value of symbol to expression
  - **.global *symbol***
    - Makes *symbol* visible outside the assembled object file
  - **.include *"filename"***
    - Provides a mechanism for including supporting files in a source program
  - **.space *size***
    - Allocates *size* bytes to be used for storage

Directives

- Commonly used directives
  - **.text**
    - Identifies the code that should be placed in the text section of memory
  - **.word *expression***
    - Expressions separated by commas are specified. Each expression is assembled into a 32-bit number

Labels

- Represents the memory address of the data or instruction marked with that label
  - **The assembler replaces each label with its memory address when generating the executable**
  - **Must start at the beginning of the line with no leading spaces**
  - **Ex:**

```
loop:     #read from r2 and store to r3
          ldbio r4, 0(r2)
          stbio r4, 0(r3)
          br    loop
```

  - **When this code is assembled, the memory address of the ldbio instruction will be inserted in the br instruction**

Registers

- NIOS II has 32 internal registers
  - **Registers hold a 32 bit value**
  - **R0 always holds 0. Writing to R0 has no effect.**
  - **R1 is for the assembler – don't use in programs**
  - **Registers r24, r25, r27, r28, r29 and r30 all have special uses by the processor**
  - **R31 holds the return address for subroutine**

Registers

Register	Name	Function	Register	Name	Function
r0	zero	0x00000000	r16		
r1	at	Assembler temporary	r17		
r2		Return value	r18		
r3		Return value	r19		
r4		Register arguments	r20		
r5		Register arguments	r21		
r6		Register arguments	r22		
r7		Register arguments	r23		
r8		Caller-saved register	r24	et	Exception temporary
r9		Caller-saved register	r25	bt	Breakpoint temporary *(1)*
r10		Caller-saved register	r26	gp	Global pointer
r11		Caller-saved register	r27	sp	Stack pointer
r12		Caller-saved register	r28	fp	Frame pointer
r13		Caller-saved register	r29	ea	Exception return address
r14		Caller-saved register	r30	ba	Breakpoint return address *(1)*
r15		Caller-saved register	r31	ra	Return address

Instruction Types

- Approximately 100 instructions
- 3 types
  - **R-type : uses 3 registers**
    - Add rC, rA, rB   rC <= rA + rB
    - Sub r2, r16, r15   r2 <= r16 - r15
    - Mul r3, r5, r6   r3 <= r5 * r6
    - Div, and, or, nor, xor, etc….

31 … 27	26 … 22	21 … 17	16 … 6	5 … 0
A	B	C	OPX	OP

Instruction Types (Con't)

- **I-type : uses 2 registers and an immediate operand**
  - Immediate operand is a 16 bit number
  - Addi rB, rA, IMMED16 rB <= rA + IMMED16
  - Subi r2, r16, -4 r2 <= r16 – (-4)
  - Muli r3, r4, 100 r3 <= r4 * 100
  - Andi, ori, xori, etc…
  - No divi

31 27	26 22	21 6	5 0
A	B	IMMED16	OP

Instruction Types (con't)

- J-Type
  - **Does not use registers**
  - **Has a 26 bit immediate operand**
  - **Immediate operand is the address of a subroutine**
  - **Only call instruction**
    - Call IMMED26
    - Call $0231F0C4
    - Call Subroutin2

31 … 6	5 … 0
IMMED26	OP

Common Instructions

- Arithmetic
  - **Add, addi**
  - **Sub, subi**
  - **Mul, muli**
  - **Div, divu (divide unsigned)**
- Logic
  - **Bitwise operations**
  - **And, andi**
  - **Or, ori**
  - **Nor, nori**
  - **Xor, xori**

R·I·T

Common Instructions (con't)

- Comparison
  - **Cmpeq rC,rA,rB   // rC <= 1 if rA = rB**
  - **Cmpne rC,rA,rB   // rC <= 1 if rA != rB**
  - **Cmplt rC,rA,rB   // rC <= 1 if rA < rB**
  - **If comparison is not true, rC <= 0**
  - **Other options for unsigned comparisons**
  - **Immediate comparisons also exist**
    - Cmpeqi rB,rA,IMMED16 //compares rA to constant
    - Cmpnei rB, rA,IMMED16
    - Etc..

Common Instructions (con't)

- Shifts
  - **Srl rC,rA,rB   rC <= rA << rB**
  - **Srli rC,rA,IMMED5 rC <= rA << IMMED5**
  - **Sra rC,rA,rB   rC <= rA >> rB**
  - **Srai rC,rA,IMMED5 rC <= rA >> IMMED5**
  - **Logical shifts fills vacated bits with 0's**
  - **Arithmetic shifts extend the sign bit on the left**
- Rotates
  - **Ror rC,rA,rB**
  - **Rol rC,rA,rB**
  - **Roli rC,rA,IMMED5**
  - **Rotate differs from a shift in that the bits shifted out of one end are wrapped around to the other end**

Common Instructions (con't)

- Unconditional Branch
  - **Br LABEL**
  - **I-type instruction. Immediate operand (16bits) is the offset from current program location to LABEL**
- Conditional Branch
  - **Blt rA,rB,LABEL //branches if rA < rB**
  - **Beq rA,rB,LABEL //branches if rA = rB**
  - **Also unsigned comparison**

R·I·T

Memory Access

source

Read: ldw r3, 4(r5) //copies value from memory to r3

Write: stw r3, 4(r5) //copies r3 value to memory

Effective address: adds constant 4 to contents of r5 to form the address

destination

Ex: If r5 contains the value 0x100, than the memory location accessed for both the ldw and stw would be 0x104.

Load and Store

- No ldi or sti (immediate load or store)
  - **How do you put 0x100 into r5?**
  - **Addi r5, r0, 0x100**
- For memory access
  - **Ldw, stw : load and store word**
  - **Ldb, stb : load and store byte**
  - **Ldh, sth : load and store half word**
- For memory mapped io
  - **Ldwio, stwio, ldbio, stbio, ….**
  - **Bypasses the cache**

Memory Structure

- Memory is organized in words
  - **32 bits or 4 bytes**
  - **Each word has an address**

Address	Data
0x0000	32 bit value
0x0004	32 bit value
0x0008	32 bit value
0x000C	:
:	:
:	:
0xFFFC	:

Why do the addresses jump by 4?

Memory Structure (con't)

- Because data is **byte addressable** memory effectively looks like this:

Address	Data
0x0000	8 bit value
0x0001	8 bit value
0x0002	8 bit value
0x0003	8 bit value
0x0004	:
:	:
0xFFFC	:

Memory Structure (con't)

Consider the following data:

Address	Data
0x0000	0x12345678
0x0004	0xABCDEF00

It is stored like this:

Address	Data
0x0000	0x78
0x0001	0x56
0x0002	0x34
0x0003	0x12
0x0004	0x00
0x0005	0xEF

LSB has smallest address:
called **little endian**

What value is put into r3 with the following Instruction, assuming r5 contains 0x0001?

Ldw r3, 4(r5)

Example code

```
/* Program that finds the sum of all the elements in an array */
/* and stores the sum in r4 */

        .text
        .equ N, (Aend-Astart)/4     #defines symbol N to hold the number of elements in array

        .global main
main:
         movi r4, 0           #initialize sum
         movi r3, N           #loop counter, N entries in A
         movia r5, Astart     #Astart is address of A
LOOP:    ldw r2, 0(r5)        #read next entry in A[]
         add r4, r4, r2
         addi r5, r5, 4       #go to next entry in A
         subi r3, r3, 1       #decrement loop counter
         bne r3, r0, LOOP     #if r3!=0, go back to LOOP
STOP:    br STOP              #endless loop to halt program

         .data
Astart:
         .word 5, 3, -6, 19, 8, 12
Aend:
         .end
```

Subroutines

- *kind of* like a function call in C
- Execution leaves the main program and jumps to a code segment that performs a specific function
- Especially useful if there is a function that needs to be done several times in a program
- Must end with ret

Subroutine Example

```
/* Program that finds the tens and ones digits of a two-digit decimal number */

        .text
        .global main
main:
        movia   r4, N
        addi    r8, r4, 4    # r8 points to the decimal digits storage location
        ldw   r4, (r4)       # r4 holds N

        call    DIVIDE       # parameter for DIVIDE is in r4
        /* Tens digit is now in r3, ones digit is in r2 */
        stb   r3, 1(r8)
        stb   r2, (r8)

END:      br      END

/* Subroutine to perform the integer division r4 / 10.
 * Returns: quotient in r3, and remainder in r2
 */
DIVIDE:   mov   r2, r4         # r2 will be the remainder
          movi  r5, 10         # divisor
          movi  r3, 0          # r3 will be the quotient
CONT:     blt   r2, r5, DIV_END
          sub   r2, r2, r5      # subtract the divisor, then ...
          addi  r3, r3, 1      # increment the quotient
          br    CONT
DIV_END:    ret                # quotient is in r3, remainder in r2

N:        .word   76             # number of entries in the list
Digits:   .space  4              # storage space for the decimal digits

        .end
```