Canvas Group Name: group\_58 Student 1: Kevin Le, Kevle@ucsc.edu, #1463503 Student 2: Matthew Wark, Mtwark@ucsc.edu

CMPE110 HA2: Human Compiler

Due Date: Friday 04/27/18

Given below is a C code snippet that computes a reduction of a vector consisting of byte sized values residing in main memory. Note the **type (uint8\_t == byte, uint32\_t == doubleword)** and size of the variables vec and result.

RISC-V Manual: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

* Vec the array and 1024 is size of array and uint8\_t is size of each block.

uint8\_t vec[1024]; //unsigned byte/char

uint32\_t result;

for (int i = 0; i < k; i++)

result += vec[i];

}

1. Transform the code snippet above into RISC-V assembly, by implementing a loop that performs k-iterations. Document every line in your assembly code, and in particular, state which registers hold which variables. If you need more than one line of documentation for one instruction, leave the next instruction table cell empty. **(4 Points)**

|  |  |  |
| --- | --- | --- |
| Label | Instruction | Documentation |
|  |  | Array memory location is in x18 |
|  |  | K memory location is in x20 |
|  | LB x5, x20, 0 | K is stored in x20 |
|  | ADDI x6, x6, 0 | Initialize the counter variable, i, in x6 |
|  | ADDI x7, x7, 0 | Initialize the results variable in x7 |
| loop\_check: | BLT x6, x5 loop\_end | If our counter is greater than or equal to K, then jump to loop\_end. Otherwise, execute loop body. |
|  | LB x28, x18, x6 | Load the byte from memory, offset by a value of i |
|  | ADDI x7, x28, x7 | Add value from element i within array to our total results |
|  | ADDI x6, x6, 1 | Increment i by 1 |
|  | JAL loop\_check | Jump to the loop condition check at loop\_check: |
| loop\_end: |  |  |
|  | SD x18, x21, 0 | Store the results back in main memory at the address in |

1. Assume k = 16. Assume a non-pipelined processor with a CPI of 8 for load and store instructions, a CPI of 2 for branches and jumps and a CPI of 1 for all other instructions. Compute the average CPI of the code. **(1 Point)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Branches/Jumps** | **Loads/Stores** | **Other** |
| Outside of loop | 0 | 2 x 8 = 16 Cycles | 2 x 1 = 2 Cycles |
| Inside of loop | 17 x 2 = 34 Cycles | 16 x 8 = 128 Cycles | 32 x 1 = 32 Cycles |

Total Cycles = 212

Total Instructions = 69

Average CPI = 212 / 69 = 3.0725 CPI

1. Based on the assumptions of 2) and a clock frequency of 2 GHz, what is the execution time of your program? **(1 Point)**

Time = Instructions \* Cycles \* Clock Time

= 69 \* 212 \* .5\*10^-9

= .000007314 seconds

1. Optimize your code from 1) by reducing the number of the **most costly** operations. Before filling out the table below, explain your proposed optimization in two sentences. **(4 Points)**
   1. The proposed solution is to reduce the number of loads needed for large values of k. We will accomplish this by taking large “chunks” of the vector at a time via a combination of Load doubleword, Load word, and load byte; using logical bit shifts to pick out values from a longer chain of values.
      1. It is also worth noting that we are doing this without nested loops. We can save a cycle or two per loop by explicitly writing out all 8/9 instructions for the “8 loop”, all 8 times, all 8/9 instructions for the “4 loop” 4 times etc. No need for using cycles on branches for looping when we know the number of iterations.

|  |  |  |
| --- | --- | --- |
| Label | Instruction | Documentation |
|  |  | Array memory location is in x1 |
|  |  | K memory location is in x2 |
|  | LB x3, x2, 0 | Load value of k from address |
|  | ADDI x4, x4, 4 | Set x4 = 4 |
|  | ADDI x8, x8, 8 | Set x8 = 8 |
|  | DIVU x5, x3, x8 | Calculate how many times the “8 loop” will run. Store in x5 |
|  | REMU x9, x3, x8 | Calculate remainder of ^ to find remaining number of loops |
|  | DIVU x6, x9, x4 | Calculate how many times the “4 loop” will run. Store in x6 |
|  | REMU x7, x3, x8 | Calculate remainder of ^ to find remaining number of loops |
|  | ADDI x10, x10, 0 | Initialize counter i to 0 |
|  | J loop\_check8 | Jump to condition check for 8 loop |
| loop\_body8: | MUL x12, x8, x10 | Multiply i \* 8 to get offset value for load |
|  | LD x15, x12(x1) | Load a doubleword, offset by increments of 8 bytes |
|  | SRLI x13, x15, 56 | Shift all but 8 bits to the right, so all but the 8 MSBs fall off |
| (Begin Rep.) | SLLI x14, x15, 8 | Shift 8 bits left so the 8-16 bits are MSBs |
|  | SRLI x14, x14, 56 | Clear all bits less significant than the 8 we want to look at |
|  | ADD x14, x13, x14 | Add the 2 bits found from shifts together |
|  | ADD x11, x11, x14 | Add value from prev addition to the results |
|  | ADDI x10, x10, 1 | Increment i |
| (End Rep.) | ... | “Rep.” repeats 8 times within this loop with the value of the left shift increasing by 8 every time. |
| loop\_check8: | BLT x10, x5, loop\_body | Reset loop if counter is still below the calculated number of runs for the 8 loop |
|  | ADD x16, x0, x12 | Holds current position in vector for further loops |
|  | SUBI x10, x10, x10 | Reset loop counter i back to 0 |
|  | J loop\_check4 | Jump to check if the condition of loop is met |
| loop\_body4: | MUL x12, x4, x10 | Multiply i \* 8 to get offset value for load |
|  | ADD x12, x16, x12 | Add current offset amount to final position from prev loop |
|  | LW x15, x12(x1) | Load a word, offset by increments of 4 bytes per loop |
|  | SRLI x13, x15, 12 | Isolate 4 most significant bits |
| (Begin Rep.) | SLLI x14, x15, 4 | Get rid of 4 MSBs, in prep to clear the other 8 LSBs |
|  | ADD x14, x13, x14 | Add the 2 bits from the shifts together and put in x14 |
|  | ADD x11, x11, x14 | Add value from prev addition to the results |
|  | ADDI x10, x10, 1 | Increment i |
| (End Rep.) | ... | Repeats 4 times within the for loop |
| loop\_check4: | BLT x10, x6, loop\_body4 | Branch back up to the start of the loop if i < k |
|  | SUBI x10, x10, X10 | Reset loop counter i back to 0 |
|  | J loop\_check | Jump to check if condition for loop is met |
| loop\_body: | LB x15, x12(x1) | Load the x10th byte from the remaining portion of vector |
|  | ADD x11, x15, x11 | Add value to result |
|  | ADDI x10, x10, 1 | Increment i |
|  | ADD x12, x12, x10 | Increment load location in vector |
| loop\_check: | BLT x10, x7, loop\_body | Reset loop if i < the number of times this loop was calculated to run |
|  | SW x11, x18, 0 | Store the results back into main memory |

1. Based on the assumptions of 2) what is the average CPI of the code in 4) now? **(1 Point)**

**K = 16**

|  |  |  |  |
| --- | --- | --- | --- |
| **Location** | **Branches/Jumps** | **Loads/Stores** | **Other** |
| Out of loops | 0 | 2 x 8 = 16 Cycles | 10 x 1 = 10 Cycles |
| 8 Loop | 2 x 2 = 4 Cycles | 2 x 8 = 16 Cycles | 82 x 1 = 82 Cycles |
| 4 Loop | 1 x 2 = 4 Cycles | 0 | 0 |
| 1 Loop | 1 x 2 = 4 Cycles | 0 | 0 |

Total Cycles: 136

Total Instructions: 100

Average CPI = 136 / 100 = 1.36 CPI

1. Based on the assumptions of 2) and a clock frequency of 2GHz, what is the execution time of the code in 4) now? **(1 Point)**

Time = Instructions \* Cycles \* Clock Time

Time = 100 \* 136 \* .5\*10^-9

= .0000068 seconds

1. Translate a recursive version of the function BitCount into RISC-V assembly code. This function counts the number of bits that are set to 1 in an integer. The parameter x is passed to your function in register x10. Your function should place the return value in register x1. Use the calling convention of RISC-V to save and restore the required registers and variables. **(8 Points)**

int BitCount(unsigned x) {

int bit;

if (x == 0)

return 0;

bit = x & 0x1; // and op between var and constant. 0x1 is just one and is a constant. 0X reps a hex.

return bit + BitCount(x >> 1); // shift to the right 10 to 01

}

|  |  |  |
| --- | --- | --- |
| Label | Instruction | Documentation |
| BitCount: | ADDI x2, x2, -32 | Allocate space on the stack |
|  | SW x18, x2, 28 | Store a saved register to allow saving of # of bits (x18) |
|  | SW x2, x2, 24 | Store the stack pointer on the stack |
|  | SW x8, x2, 20 | Store the frame pointer on the stack |
|  | ADDI x8, x2, 28 | Set the new position for the frame pointer |
|  | ADD x6, x10, x0 | Set value of x in this function call |
|  | J base\_check | Jump to the condition check for the base case |
| base: | ADDI x9, x0, 0 | Load 0 from x0 into the return value |
|  | J exit | Jump to the exit label to restore from the stack return to RA |
| base\_check: | BEQ x6, x0, exit | If x == 0, return |
|  | ANDI x8, x5, 1 | Bitmask with 1 to find if the LSB is a 1 |
|  | SRLI x6, x6, 1 | Shift bits to the right by 1 to look at next col in next call |
|  | ADDI x10, x6, 0 | Set argument for next call to the val of x after shifting it |
|  | SW x10, x2, 16 | Store function arguments on stack |
|  | JAL x1, BitCount | Recursive call to the program |
|  | ADD x1, x1, x8 | Add value returned from recursive call to current # of bits |
| exit: | LW x10, x2, 16 | Step 6.5, restore from stack |
|  | LW x18, x2, 28 | Restore the saved register from the stack |
|  | LW x9, x2, 20 | Restore the return address from the stack |
|  | LW x8, x2, 24 | Restore the frame pointer from the stack |
|  | ADDI x2, x2, 32 | Return memory used for the stack to the system |
|  | JALR x9 | Return to RA |