Development Phase Benchmarks

Prepared by:

JoSDC'24 Mentors Team

This file contains a set of carefully designed benchmarks to help you test, verify, and push your designs. As mentors, our goal is to support you through this journey, ensuring you gain confidence in your project and its performance. Here are a few tips and guidelines to make the best use of these benchmarks:

- Each benchmark tests specific sets of instructions or scenarios, designed to highlight different functionalities and potential edge cases in your design.
- Take time to understand what each benchmark is testing. Look for specific instructions or patterns that may challenge your design and observe how your implementation handles these cases.
- We're here to help! Don't hesitate to ask questions, whether it's about interpreting results, resolving issues, or understanding benchmark details. You can contact through the Q&A channel. Also, feel free to submit a meeting request if you want more help, using following link : "JoSDC'24 Meetings Form".
- Check out the additional documentation and training sessions provided throughout the development phase. These are designed to complement your work with the benchmarks and help you improve your design.

Additional Important Notes and Advice for Testing:

- **Committee Evaluation of Benchmarks:** These benchmarks will be assessed by the committee alongside any additional benchmarks you provide. Ensuring comprehensive and well-documented results will aid in your project's evaluation.
- **Testing all benchmarks is preferred;** however, if constraints prevent you from testing the entire set, it's acceptable to test a portion. Clearly document any untested benchmarks to help the committee understand your approach and limitations.
- **Showcase Your Testing and Tools:** Demonstrate how you've used the software tools developed in this phase, such as your assembler and cycle-accurate simulator, in handling the benchmarks. If any aspects of your tools meet only the minimum requirements, it's acceptable to manually assist the tool in completing the benchmarks.
- **Execution Details:** Provide execution details for each benchmark, including whether functionality passed or failed, along with performance metrics such as cycles, execution time, instruction count, flushed instructions, stalls, etc. This data will be critical for evaluating the efficiency and accuracy of your design.
- **Document Bug Discovery and Resolution:** If the benchmarks reveal any bugs in your design, outline the steps for reproducing these issues and explain the steps you took to resolve them. This demonstrates your debugging process and problem-solving skills.
- **Record Bug Findings in Documentation:** It's highly recommended to document any bugs discovered during testing, with steps for reproduction, in your final documentation. This not only aids the committee in evaluating your design but also shows a comprehensive approach to refining and verifying your work.

Good Luck, and Wish You the Best 😊

Benchmark 1 – Data Manipulation Instructions

Benchmark Description

- Benchmark author: Abdalrahman Alshannaq
- Simple benchmark to test data manipulation instructions (without branch and jump instructions included)
- Minimal percentage of dependencies to test general functionality of each instruction

Benchmark Code

```
    .data
value: .word 0x5                    # sample data for loading

    .text
main:
    # Immediate instructions to initialize registers
    ADDI $1, $0, 0xA
    ORI  $2, $0, 0xB
    XORI $3, $0, 0xC

    # Basic ALU operations
    ADD  $4, $1, $2
    SUB  $5, $4, $3
    AND  $6, $1, $3
    OR   $7, $2, $3
    NOR  $8, $2, $3
    XOR  $9, $1, $2

    # Comparison operations
    SLT  $10, $1, $2
    SGT  $11, $3, $1

    # Shift operations
    SLL  $12, $1, 2
    SRL  $13, $2, 1

    # Load and store word instructions
    LW   $14, 0x0($0)
    SW   $4, 0x0($0)
```

Benchmark 2 – Control Flow Instructions

Benchmark Description

- Benchmark author: Issa Qandah
- Simple benchmark to test control flow instructions (‘BEQ’, ‘BNE’, ‘BLTZ’, ‘BGEZ’, ‘JAL’)
- Minimal dependencies to test the functionality of each control flow instruction.

Benchmark Code

```
        .text
main:
                # Initialize registers with immediate values
                ADDI $1, $0, -5
                ADDI $2, $0, 5
                ADDI $3, $0, 5
                ADDI $4, $0, 84

L1:
                #  Test BEQ
                BEQ $2, $3, L2
                NOP           # No operation / SLL $0, $0, 0
                JAL L1
L2:
                # Test BNE
                BNE $2, $3, L3
L3:
                NOP
L4:
                # Test BLTZ
                BLTZ $1, L5
                NOP
                JAL L4
L5:
                #  Test BGEZ
                BGEZ $2, L6
                NOP
                JAL L5
L6:
                #  Test JAL
                JAL L7
                NOP
                JAL L6
L7:
                #  Test JR
                JR $4
                NOP
                JAL L7
L8:
                NOP              # end of test cases
```

Benchmark 3 – Sum of Numbers

Benchmark Description

- Benchmark authors: Abdullah Matar & Yara Altamimi
- Simple benchmark to find sum of numbers from 0 to 10
- Complexity is O(N), N=10.
- Benchmark is scalable, by changing initialization value of $4

Benchmark Code

```
    .text
main:
    # Initialize registers
    ORI $2, $0, 0x1
    ORI $3, $0, 0x0
    ORI $4, $0, 0xA

SUM_LOOP:
    ADD  $3, $3, $2
    ADDI $2, $2, 1
    SLT  $5, $4, $2
    BEQ  $5, $0, SUM_LOOP

    SW   $3, 0x0($0)

END:
    NOP                     # End program
```

Benchmark 4 – Binary Search

Benchmark Description

- Benchmark author: Abdalrahman Ebdah
- Binary Search algorithm
- Contain a while loop and use multiple comparisons to break the loop
- Perform integer division using SLR instruction to calculate the mid pointer
- Benchmark Complexity: O(logn)
- Support both Word and Byte Addressing

Binary Search Algorithm

```
while (right <= left) {
        mid = (right + left)/2;
        if(a[mid] == searchElement)
                answer = mid;
                break;

        else if(a[mid] > searchElement)
                left = mid+1;

        else
                right = mid-1;
}
```

Benchmark Code

```
    .data
myArray:   .word 1, 4, 5, 7, 9, 12, 15, 17, 20, 21, 30
arraySize: .word 11

    .text
#Initailiztion
addi $1, $0, 0x0
addi $2, $0, 0xB
addi $3, $0, 0x7

loop:
    slt $7, $2, $1
    bne $0, $7, notFound
    add $4, $2, $1
    srl $5, $4, 1

    #sll $5, $5, 2 For byte addressable memory

    lw  $6, 0x0($5)

    #srl $5, $5, 2 For byte addressable memory

    beq $3, $6, found
    slt $6, $6, $3
    beq $6, $0, leftHalf
    j   rightHalf

leftHalf:
    add $2, $5, 0xFFFF      # "FFFF=-1"
    j   loop

rightHalf:
    addi $1, $5, 0x1
    j    loop

found:
    add $8, $0, $5
    j finish

notFound:
    addi $8, $0, 0xFFFF
    j finish

finish:
    NOP
```

Benchmark 5 – Max and Min in Array

Benchmark Description

- Benchmark author: Hassan TaqiEddin
- Find the maximum and minimum value of array.
- Contains one loop and tested variety of instructions.

Benchmark Algorithm

```
for i <- 1 to size - 1
    do key <- Array[i]
       if (key > max)
              max = key
              continue
       if (key < min)
              min = key
```

Registers and memory used in implementation

```
$2 : i (loop index)
$5 : Temporary register for calculating array offsets (address for
Array[i])

$10 : max (holds the maximum value)
$15 : min (holds the minimum value)
$16 : Temporary register for the value of Array[i]
$20 : size (size of the Array)

$25 : temp for condition
$26 : temp for condition
$27 : temp for condition
```

Benchmark Code

```
    .data
Array: .word 0x10, 0xF, 0x5, 0x9, 0x20, 0x19, 0x4, 0x1E, 0x9, 0xB

    .text
main:
    # Initialize registers
    ORI  $2 , $0, 0x0
    ADDI $20, $0, 0xA
    XORI $31, $0, 0x1
    ANDI $5 , $0, 0x0
    LW   $10, 0x0($5)
    LW   $15, 0x0($5)

LOOP:
    ADDI $2,  $2, 1
    SGT  $25, $20, $2
    BNE  $25, $31, END

    # Choose one of these Insertion based on your memory
    # For Word addressable         # For byte addressable
    # ADD  $5, $2, $0              # SLL  $5, $2, 2

    LW   $16, 0x0($5)
    SGT  $26, $16, $10
    BEQ  $26, $0, MIN
    OR   $10, $16, $0
    J    LOOP

MIN:
    SLT  $27, $16, $15
    BEQ  $27, $0, LOOP
    ADD  $15, $16, $0
    J    LOOP

END:
    NOP # (NOP equals to SLL $0, $0, 0)
```

Additional Notes

- Benchmark complexity: O(N).
- Benchmark is scalable by changing the initialization value of $20 and adding memory elements.
- Both Word and Byte addressing supported, choose memory addressing by using the commented instructions as needed.

Benchmark 6 – Insertion Sort

Benchmark Description

- Benchmark author: Abdalrahman Alshannaq.
- Insertion Sort algorithm implementation.
- Contain two nested loops and multiple branches and jumps.
- Test multiple cases and targeting different dependencies.

Insertion Sort Algorithm

```
for i <- 1 to size - 1
// (assuming first index on location 0, and last index on location size-1)
     do key <- Arr[i]
        j <- i - 1
        while j > -1 and Arr[j] > key
             do Arr[j+1] <- Arr[j]
                j <- j-1
        Arr[j+1] <- key
```

Registers and memory used in implementation

```
$1  : i
$2  : j
$10 : key
$20 : size of array

$5  : temp as address for i
$6  : temp as address for j
$7  : temp as address for j+1
$15 : temp as value of Arr[i]
$16 : temp as value of Arr[j]
$17 : temp as value of Arr[j+1]

$22 : holds -1

$25 : temp for condition
$26 : temp for condition
$27 : temp for condition
$28 : temp for condition

Memory used: first 10 elements (words) in data memory.
```

Benchmark Code

```
     .data
Arr: .word 0x5, 0x7, 0x2, 0xF, 0xA, 0x10, 0x30, 0x1, 0xFF, 0x55

     .text
main:
               # Initialize registers
               ORI  $2, $0, 0x0
               XORI $10, $0, 0x0
               ADDI $20, $0, 0xA
               ADD  $5, $0, $0
               ADDI $1, $0, 0x1
               ADDI $22, $0, -1
LOOP1:
               # ADD  $5, $1, $0       # Word Addressing mode
               # SLL $5, $1, 2         # Byte Addressing mode
               LW   $15, 0x0($5)
               OR   $10, $15, $0
               ADDI $2, $1, -1
LOOP2:
               # use one line according to your addressing mode
               # ADD  $6, $2, $0       # Word Addressing mode
               # SLL $6, $2, 2         # Byte Addressing mode
               LW   $16, 0x0($6)
               SGT  $25, $2, $22
               SGT  $26, $16, $10
               AND  $27, $26, $25
               BEQ  $27, $0, EXIT2
               ADDI $7, $2, 0x1
               # this line is commented, you may use it only if your memory is byte addressable
               # SLL $7, $7, 2
               SW   $16, 0x0($7)
               ADDI $2, $2, -1
               J  LOOP2
EXIT2:
               ADDI $7, $2, 0x1
               # this line is commented, you may use it only if your memory is byte addressable
               # SLL $7, $7, 2
               SW $10, 0x0($7)
               ADDI $1, $1, 0x1
               SLT  $28, $1, $20
               BNE  $28, $0, LOOP1
FINISH:
               NOP # (NOP equals to SLL $0, $0, 0)
```

Additional Notes

- Benchmark complexity: $O(N^2)$.
- Benchmark is scalable by changing the initialization value of $20 and adding memory elements.
- Both Word and Byte addressing supported, word addressing is default case, change to byte memory addressing by using the commented instructions as needed.