CSCE 5610: Computer Architecture

Due date extension for HW -1 until Thursday
You can hand me hardcopies, or email them to me and the grader

Hints for HW 1.
1.3 is an extension of 1.1 and 1.2
Phoenix is a 200 sq.mm chip and has room for 8 cores
So, we are asked assume that each core will occupy 25 sq. mm or 0.25 sq. cm
Using this area, compute yield
= 1/[1+(0.04*0.25)] = 0.87

Now we need to use probability theory to obtain yields for 1, 2, 8 good cores
(# possible combinations of N cores)*(yield)^N*(1-yield)^N

<table>
<thead>
<tr>
<th># defect-free</th>
<th># combinations</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0.32522167</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0.39234892</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>0.20519192</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
<td>0.06132172</td>
</tr>
<tr>
<td>4</td>
<td>70</td>
<td>0.01145377</td>
</tr>
<tr>
<td>5</td>
<td>56</td>
<td>0.00136919</td>
</tr>
<tr>
<td>6</td>
<td>28</td>
<td>0.00010231</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>4.3673E-06</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>8.1573E-08</td>
</tr>
</tbody>
</table>

Phoenix^1 = yield for 1 = 0.00000436
Phoenix^2 = (yield for 2 or 3)
Phoenix^4 = (yield for 4, 5, 6 or 7)

1.5
a). Application of Amdahl’s law. Find out the speedup achieved.
Instead of using 4 cores, can we achieve the same speedup using one core
at a higher frequency (speed is proportional to frequency)
b). Compare the energy using 4 cores at original frequency with 1 core at faster clock
c). Here we are comparing energy (and power) using 4 cores vs 2 cores and 2 ASICs
⇒ turn of 2 cores completely and ASIC consumes 20% of the power

1.9. Is straightforward
1.12. Apply Amdahl’s law
CSCE 5610: Computer Architecture

Review:

Instruction set choices

Number of operands (0, 1, 2, ..)

Register-Register; Register-Memory, Memory-Memory

Load/Store (all computations on register values)

Big endian vs Little Endian

Memory addresses

Data alignment

Specifying addresses → address modes

Register direct, Register indirect, Immediate

Register+displacement (or indexed)

memory direct and memory indirect

Most common is the Register+displacement

LD R<destination>, displacement(R<index>)

© 2019 Elsevier Inc. All rights reserved.

Figure A.6 Selection of addressing modes with examples, meaning, and usage.

In autoincrement/-decrement and scaled addressing modes, the variable d designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode.

The extensions to C used as hardware descriptions are defined on page A.38.
Displacement can be accommodated in 12-16 bit fields
Same is true for immediate (constant) values or branch-addresses (PC relative)

How many registers? 32 or 64
   if we use 64, we need 6 bits to specify registers as operands
   if we use 32 registers we need 5 bits
   Separate Integer and Floating point registers
   64 of each kind?
      implied by operations (Integer-Add means use integer registers)

Special purpose registers (PC, Status register, Stack-pointer ?)
   Some systems use general purpose (integer) registers as SP, Frame Pointer even as PC

Most machines use 32 integer, single and double precision registers (32 bits and 64 bits)
   In some cases Integer register zero always contains zero

What operations should we support?
   Arithmetic – integer and floating point
      Add, Subtract, Multiply and Divide
   Logic (bit, byte or word)
      AND, OR, NOT, …
   Compare: EQ, NE, GE, LT,…
   Conditional and unconditional branches
   Delayed branch?

Function call:  RISC V or ARM style: JAL – jump and link (or Branch and Link)
   PC is saved in R31 ($ra or return address register)
   By convention you can pass some parameters through registers
      R4-R7 ($a0-$a3 or arguments registers)
   Results can also be returned through registers
      R2-R3 ($v0-$v1)
   Callee Saves registers R16-R23 ($s or save registers)
   Caller Saves registers R8-R15 ($t or temporary registers)
Register Windows -- Sun SPARC

Overlapped registers
Global registers

Support only commonly used instructions -- need to study in more detail
What type of ALU operations: multiply, divide, sq. rt.
CSCE 5610/5610: Computer Architecture

A single instruction computer -- ULTIMATE RISC
Subtract and Branch if negative
SBN, R1, R2, Offset

SBN R3, R3, +1  set R3 to zero
SBN R3, R2, +1  set R3 to -(R2)
SBN R1, R3, +1  set R1 to R1+R2

Instructions without a program counter

Option 1: The address of the next instruction is included in the instruction itself.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op</th>
<th>R2</th>
<th>R4</th>
<th>R5</th>
<th>Next Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td>R2</td>
<td>R4</td>
<td>R5</td>
<td>2001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op</th>
<th>R2</th>
<th>R4</th>
<th>R5</th>
<th>Next Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td></td>
<td>R2</td>
<td>R4</td>
<td>R5</td>
<td>2038/2001</td>
</tr>
</tbody>
</table>

CSCE 5610: Computer Architecture

Instructions without a program counter

Option 2: Dataflow instructions

### MIPS like instructions

1. LOAD R2, A / load A into R2
2. LOAD R3, B / load B into R3
3. ADD R11, R2, R3 / R11 = A+B
4. LOAD R4, X / load X into R4
5. LOAD R5, Y / load Y into R5
6. ADD R10, R4, R5 / R10 = X+Y
7. SUB R12, R4, R5 / R12 = X-Y
8. MULT R14, R10, R11 / R14 = (X+Y)*(A+B)
9. DIV R15, R12, R11 / R15 = (X-Y)/(A+B)
10. STORE ..., R14 / store first result
11. STORE ..., R15 / store second result

### Pure Dataflow Instructions

1. LOAD 3L / load A, send to Instruction 3
2. LOAD 3R / load B, send to Instruction 3
3. ADD 8R, 9R / A+B, send to Instructions 8 and 9
4. LOAD 6L, 7L / load X, send to Instructions 6 and 7
5. LOAD 6R, 7R / load Y, send to Instructions 6 and 7
6. ADD 8L / X+Y, send to Instructions 8
7. SUB 9L / X-Y, send to Instruction 9
8. MULT 10L / (X+Y)*(A+B), send to Instruction 10
9. DIV 11L / (X-Y)/(A+B), send to Instruction 11
10. STORE / store first result
11. STORE / store second result
CSCE 5610: Computer Architecture

For the rest of the class we will assume that we are using RISC V instructions
See Appendix Pages A-33 to A-42

All instructions are 32 bits long
There are 32 integer and 32 floating point registers
Displacements and branch target fields are 12 bits
There are 3 instruction types or formats (but bits are numbered backward)

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>func7</td>
<td>rs2</td>
<td>rs1</td>
<td>func3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>func3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type
I-type
S-type
U-type

Note the actual instruction is specified by opcode + func3+func7 fields
For example all ALU instructions use the same opcode and then the func3 indicates if it is an ADD, SUB etc

CSCE 5610: Computer Architecture

<table>
<thead>
<tr>
<th>Instruction format</th>
<th>Primary use</th>
<th>rd</th>
<th>rs1</th>
<th>rs2</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Register-register ALU instructions</td>
<td>Destination</td>
<td>First source</td>
<td>Second source</td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td>Load</td>
<td>Destination</td>
<td>First source base register</td>
<td>Value displacement</td>
<td></td>
</tr>
<tr>
<td>S-type</td>
<td>Store and branch</td>
<td>Base register first source</td>
<td>Data source to store second source</td>
<td>Displacement offset</td>
<td></td>
</tr>
<tr>
<td>U-type</td>
<td>Jump and link</td>
<td>Register destination for return PC</td>
<td>Target address for jump and link</td>
<td>Target address for jump and link</td>
<td></td>
</tr>
</tbody>
</table>

CSCE 5610 Sept. 11, 2018
Figure A.25 The load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign-extended and zero-extended forms. All memory references use a single addressing mode. Of course, both loads and stores are available for all the data types shown. Because RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

There are 32 integer registers and labeled as x0…x31
There are 32 single precision registers f0,…f31 and 32 double precision d0,…d31
Register x0 is always zero

Control flow related instructions

Example instruction Instruction name Meaning
jal x1,offset Jump and link Regs[x1] = PC + 4; PC = Regs[x1] + offset
jalr x1,x2,offset Jump and link register Regs[x1] = PC + 4; PC = Regs[x1] + offset
beq x3,x4,offset Branch equal zero if (Reg[x3] == Reg[x4]) PC = PC + (offset << 1)
bgt x3,x4,offset Branch not equal zero if (Reg[x3] > Reg[x4]) PC = PC + (offset << 1)
Figure A.28 A list of the vast majority of instructions in RV64G.
This list can also be found on the back inside cover. This table omits system instructions, synchronization and atomic instructions, configuration instructions, instructions to reset and access performance counters, about 10 instructions in total.

Problem A.7. Convert the following C code into RISC V instructions

```c
for (i=0; i<100; i++) A[i] = B[i] + C;
```

Let us assume the starting address of A (that is A[0]) is in x1 and that of B is in x2
And the address of C is in x3. We will assume 64-bit integers.

Integer registers are 64-bit long
```
ADD  x4, x0, x0 ; initialize i=0 in x4
ADD x9, x0, #100 ; x9=100
Loop:
SLT x5, x4, #3 ; x5 = 8*i
ADD x5, x5, x2 ; x5 contains the address of B[i]
LD x8, (x5) ; x8 = B[i]
ADD x8, x8, x3 ; x8 = B[i]+C
SLT x6, x4, #3 ; x6 = 8*i
ADD x6, x6, x1 ; x6 contains the address of A[i]
SD x8, (x6) ; store x8 at A[i]
ADD x4, x4, #1 ; i = i+1
BLT x4, x9, loop ; if i<100 go to loop
```
But now we will start on memory systems (Chapter 2)
Some basics.
Your programs (in C or other) are translated by compilers
The executable is basically your instructions and static data
Each instruction will have a memory address where it will be stored
Each static data item will have an address when compiled
dynamic data (malloc, new) will be assigned addresses at run time

*So, each instruction and data item (variable) will have an address*

- instruction addresses are in the program counter; instructions are 4 bytes long
- data addresses are generated using Load and Store instructions

These addresses are “Virtual” addresses which will be mapped physical addresses

Each address is either 32 or 64 bits and refer to a byte
for 4 byte data type (integer), your byte address will look like
xxxx........xxx 00
Or addresses will be on 4 byte boundary

---

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    int sum = 0;
    for (i = 0; i < 100; i++)
        sum += i;
    printf("The sum from 0..100 is %d\n", sum);
    return 0;
}
```

---

```c
int main (int argc, char *argv[]) {
    int i;
    int sum = 0;
    for (i = 0; i < 100; i++)
        sum += i;
    printf("The sum from 0..100 is %d\n", sum);
    return 0;
}
```
Dynamic data relates to malloc, new, etc

All these addresses are “virtual”
At runtime, physical pages are allocated for the virtual pages

So we need to understand the difference between a virtual and physical address
Addresses generated by your program (either instruction address, or address generated by a Load or Store) are virtual addresses