NYU Tandon School of Engineering

Fall 2025, ECE 6913

Review Problems & Solutions

1. The results of the SPEC CPU2006 bzip2 benchmark running on an AMD Barcelona has an instruction count of 2.389E12, an execution time of 750 s, and a reference time of 9650 s

a. Find the CPI if the clock cycle time is 0.333 ns.

IC = 2.389T, ET=750s, RefET = 9650s

CPI = (ET * clock rate)/IC = 750s x 3.00GHz / 2.389T = **0.94**

b. Find the SPECratio

SPECratio = T_ref/T = 9650s /750s = **12.86**

c. Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% without affecting the CPI

ET = CPI x IC x Tcycle = 0.94 x 2.389T x 1.1 x 0.333ns = 822s or an **increase in ET of 9.67%**

d. Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% and the CPI is increased by 5%.

ET = CPI x IC x Tcycle = 0.94 x 1.05 x 2.389T x 1.1 x 0.333ns = 863s or an **increase in ET of 15%**

e. Find the change in the SPECratio for this change

SPECratio = T_ref/T = 9650s /863s = **11.18**

f. Suppose that we are developing a new version of the AMD Barcelona processor with a **4 GHz clock rate**. We have added some additional instructions to the instruction set in such a way that *the number of instructions has been reduced by 15%.* The *execution time is reduced to 700 s* and the new SPECratio is 13.7. Find the new CPI.

ETnew = 700s = CPIX x 2.389T x 0.85 x 0.250ns

So, CPIX = 700/507.66 = **1.38**

g. This CPI value is larger than obtained in part a. as the clock rate was increased from 3 GHz to 4 GHz. Determine whether the increase in the CPI is similar to that of the clock rate. If they are dissimilar, why?

CPI increased by 1.38/0.94 = 1.46

Clock rate increased by 4/3 = 1.33

Both – Instruction Count *and* the Clock Cycle time were lowered as a result of adding instructions to the instruction set.

Since each instruction, on average, is accomplishing more work, the reduced cycle time is likely to require the average instruction to complete its task using more cycles and hence an increase in the average CPI of the Benchmark

h. By how much has the CPU time been reduced?

700s / 750s = 0.933 or **by 6.67%**

2. Assume a program requires the execution of 50 $\times 10^6$ FP instructions, 110 $\times 10^6$ INT instructions, 80 $\times 10^6$ L/S instructions, and 16 $\times 10^6$ branch instructions. The CPI for each type of instruction is 1, 1, 4, and 2, respectively. Assume that the processor has a 2 GHz clock rate.

a. By how much must we improve the CPI of FP instructions if we want the program to run two times faster?

The total number of clock cycles for the program is given by:
Total Clock Cycles = (CPIfp × #FP instructions) + (CPIint × #INT instructions) + (CPIl/s × #L/S instructions) + (CPIbranch × #branch instructions)

Substituting the values given:

Total Clock Cycles = (1 × 50 × 10^6) + (1 × 110 × 10^6) + (4 × 80 × 10^6) + (2 × 16 × 10^6)
= 50 × 10^6 + 110 × 10^6 + 320 × 10^6 + 32 × 10^6 = 512 × 10^6 cycles

To run the program twice as fast, we need to reduce the total clock cycles to:

New Total Clock Cycles = 512 × 10^6 / 2 = 256 × 10^6 cycles

If we improve only the FP instructions, we would still need to account for the other instructions (INT, L/S, and Branch), which use a total of:

Non-FP Clock Cycles = (1 × 110 × 10^6) + (4 × 80 × 10^6) + (2 × 16 × 10^6) = 110 × 10^6 + 320 × 10^6 + 32 × 10^6 = 462 × 10^6 cycles

This exceeds the target of 256 million cycles, even without considering FP instructions. Therefore, improving the CPI of FP instructions alone cannot reduce the total clock cycles by half. Hence, the program cannot run two times faster by improving the CPI of FP instructions alone.

b. By how much must we improve the CPI of L/S instructions if we want the program to run two times faster?

Now, let's consider improving the CPI of Load/Store (L/S) instructions.

The number of clock cycles consumed by FP, INT, and Branch instructions is:

Non-L/S Clock Cycles = (1 × 50 × 10^6) + (1 × 110 × 10^6) + (2 × 16 × 10^6) = 50 × 10^6 + 110 × 10^6 + 32 × 10^6 = 192 × 10^6 cycles

To run the program twice as fast, we need the total clock cycles to be 256 million. So, the remaining cycles allocated for L/S instructions are:

L/S Clock Cycles = 256 × 10^6 - 192 × 10^6 = 64 × 10^6 cycles

Given that the number of L/S instructions is 80 × 10^6, we need the CPI of L/S to be:

CPI_{L/S} = 64 × 10^6 / 80 × 10^6 = 0.8

Thus, the CPI of L/S instructions must be reduced to 0.8 to achieve a two-fold speedup.

c. By how much is the execution time of the program improved if the CPI of INT and FP instructions is reduced by 40% and the CPI of L/S and Branch is reduced by 30%?

The initial number of clock cycles is:

Initial Clock Cycles = (1 × 50 × 10^6) + (1 × 110 × 10^6) + (4 × 80 × 10^6) + (2 × 16 × 10^6)

= 512 × 10^6 cycles

Now, apply the CPI reductions:
- INT and FP are reduced by 40%, so the new CPI is 0.6 for both.
- L/S and Branch are reduced by 30%, so the new CPI is 2.8 for L/S and 1.4 for Branch.

The new number of clock cycles is:
New Clock Cycles = (0.6 × 50 × 10^6) + (0.6 × 110 × 10^6) + (2.8 × 80 × 10^6) + (1.4 × 16 × 10^6)
New Clock Cycles = 30 × 10^6 + 66 × 10^6 + 224 × 10^6 + 22.4 × 10^6

= 404.4 × 10^6 cycles

Speedup = 512 × 10^6 / 404.4 × 10^6 ≈ 1.266

This corresponds to a 26.6% improvement in execution time.

3. Instruction(s) to copy contents at one memory location to another: B[g] = A[i+j-3]
Assume i, j, g values are in registers x5, x6, x7. Assume base address in memory of Array data structures 'A, B ' (or address in memory of 'A[0]' and 'B[0]') are stored in Registers x28, x29

```
# Add i and j (i = x5, j = x6)
add x5, x5, x6          # x5 = i + j

# Subtract 3 from the result
addi x5, x5, -3         # x5 = i + j - 3

# Multiply the result by 4 (for word address calculation)
slli x5, x5, 2          # x5 = 4 * (i + j - 3)

# Add the base address of array A (A[0] is in x28)
add x5, x5, x28         # x5 = address of A[i + j - 3]

# Multiply g by 4 (for word address calculation, g is in x7)
slli x7, x7, 2          # x7 = 4 * g

# Add the base address of array B (B[0] is in x29)
add x7, x7, x29         # x7 = address of B[g]


# Load word from A[i + j - 3]
lw x5, 0(x5)            # Load A[i + j - 3] into x5

# Store word into B[g]
sw x5, 0(x7)            # Store value into B[g]
```

4. In the next 3x problems, assume that the variables
f, g, h, i, and j are assigned to registers x5, x6, x7, x28, x29 respectively. Assume base addresses in memory of Array data structures 'A, B' (or addresses in memory of 'A[0]', 'B[0]' and 'C[0]') are stored in Registers x27, x30,x31.
Write RISCV code that implements

a. A[i] = B[2i+1], C[i] = B[2i]

```
# Calculate 2i + 1
add x26, x28, x28       # x26 = 2 * i
addi x26, x26, 1        # x26 = 2 * i + 1
slli x26, x26, 2        # Multiply by 4 (byte addressing)
add x26, x26, x30       # x26 = address of B[2i+1]

# Calculate A[i]
slli x25, x28, 2       # Multiply i by 4 (byte addressing)
add x25, x25, x27      # x25 = address of A[i]

# A[i] = B[2i+1]
lw x24, 0(x26)        # Load B[2i+1] into x24
sw x24, 0(x25)        # Store x24 into A[i]
```

b. A[i] = 2B[i-1] + 4C[i+1]

```
# Calculate B[i-1]
addi x26, x28, -1      # x26 = i - 1
slli x26, x26, 2       # Multiply by 4 (byte addressing)
add x26, x26, x30      # x26 = address of B[i-1]

# Calculate C[i+1]
addi x25, x28, 1      # x25 = i + 1
slli x25, x25, 2        # Multiply by 4 (byte addressing)
add x25, x25, x31      # x25 = address of C[i+1]

# Calculate A[i]
slli x24, x28, 2        # Multiply i by 4 (byte addressing)
add x24, x24, x27      # x24 = address of A[i]

# Load values and compute
```

```
lw x23, 0(x26)         # Load B[i-1] into x23
slli x23, x23, 1       # Multiply B[i-1] by 2
lw x22, 0(x25)         # Load C[i+1] into x22
slli x22, x22, 2       # Multiply C[i+1] by 4
add x23, x23, x22      # 2B[i-1] + 4C[i+1]

# Store result in A[i]
sw x23, 0(x24)       # A[i] = 2B[i-1] + 4C[i+1]
```

c. f = g - A[C[8] + B[4]]

```
# Load B[4] and C[8]
lw x30, 16(x30)      # Load B[4] into x30
lw x31, 32(x31)      # Load C[8] into x31

# Calculate C[8] + B[4] and address of A[C[8] + B[4]]
add x24, x30, x31      # x24 = C[8] + B[4]
slli x24, x24, 2       # Multiply by 4 (byte addressing)
add x4, x27, x24       # x4 = address of A[C[8] + B[4]]

# Load A[C[8] + B[4]] and compute f
lw x4, 0(x4)          # Load A[C[8] + B[4]] into x4
sub x5, x6, x4        # f = g - A[C[8] + B[4]]
```

5. Assume that for a given program 70% of the executed instructions are arithmetic, 10% are load/store, and 20% are branch.

a. Given this instruction mix and the assumption that an arithmetic instruction requires two cycles, a load/store instruction takes six cycles, and a branch instruction takes three cycles, find the average CPI.

CPI of program = weighted sum of instruction mix

Given an instruction mix with 70% arithmetic, 10% load/store, and 20% branch, we need to find the average CPI. Each instruction type takes a different number of cycles (arithmetic = 2, load/store = 6, branch = 3).

To calculate the average CPI:
CPI = (0.7 * 2) + (0.1 * 6) + (0.2 * 3)

After solving, we get:
CPI = 2.6

b. For a 25% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

To achieve a 25% performance improvement, the CPI must decrease since performance is inversely proportional to CPI. We reduce the CPI by 25%, so the new CPI becomes:
New CPI = 2.6 * 0.75 = 1.95

Now, only the arithmetic instructions can be optimized. We set up the equation where the cycles for arithmetic instructions are x, and load/store and branch cycles remain the same:
0.7 * x + 0.1 * 6 + 0.2 * 3 <= 1.95

After solving, we find:
x <= 1.07

c. For a 50% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

For a 50% performance improvement, the CPI needs to be halved:
New CPI = 2.6 * 0.5 = 1.3

Again, only arithmetic instructions are improved, so we modify the equation:
0.7 * x + 0.1 * 6 + 0.2 * 3 <= 1.3

After solving, we get:
x <= 0.14

6. Assume for a given processor the CPI of *arithmetic* instructions is 1, the CPI of *load/store* instructions is 10, and the CPI of *branch* instructions is 3. Assume a program has the following instruction breakdowns: 500 million *arithmetic* instructions, 300 million *load/store* instructions, 100 million *branch* instructions.

a. Suppose that new, more powerful *arithmetic* instructions are added to the instruction set. On average, through the use of these more powerful *arithmetic* instructions, we can *reduce the number* of *arithmetic* instructions needed to execute a program by 25%, while *increasing the clock cycle time* by only 10%. Is this a good design choice? Why?

We need to evaluate whether adding more powerful arithmetic instructions (reducing the number of arithmetic instructions by 25%) while increasing the clock cycle time by 10% is a good design choice.

Current CPU Cycles:

- Arithmetic: 500 million * 1 cycle

- Load/Store: 300 million * 10 cycles
- Branch: 100 million * 3 cycles

The total cycle count for the current CPU is:
Cycles = 500 * 1 + 300 * 10 + 100 * 3 = 3800 cycles

New CPU Cycles:

- Arithmetic instructions reduced by 25%, so 0.75 * 500 = 375 million instructions.
- The total cycle count for the new CPU is:
  New Cycles = 375 * 1 + 300 * 10 + 100 * 3 = 3675 cycles

Since the new CPU's clock cycle time is 10% longer, we multiply the new cycle count by 1.10:
Adjusted Cycles = 3675 * 1.10 = 4042.5 cycles

Thus, the new CPU would take longer than the original CPU (4042.5 cycles vs. 3800 cycles), making it a bad design choice.

b. Suppose that we find a way to double the performance of arithmetic instructions. What is the overall speedup of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times?

Now, calculate the speedup if the performance of arithmetic instructions is doubled or improved by a factor of 10.

1. Doubling Arithmetic Performance:
   Doubling the performance reduces the CPI for arithmetic instructions to 0.5. The new cycle count is:
   Cycles = (500 * 0.5) + (300 * 10) + (100 * 3) = 3550 cycles

The speedup is the ratio of the original cycle count to the new cycle count:
Speedup = 3800 / 3550 ≈ 1.07

2. Improving Arithmetic Performance by 10x:
   If the performance of arithmetic instructions is improved by a factor of 10, the CPI for arithmetic instructions becomes 0.1. The new cycle count is:
   Cycles = (500 * 0.1) + (300 * 10) + (100 * 3) = 3350 cycles

The speedup is:
Speedup = 3800 / 3350 ≈ 1.13

These improvements show that improving arithmetic instructions results in moderate gains in overall performance.

7 a. Provide the instruction type and assembly language instruction for the following binary value:

0000 0000 0001 0000 1000 0000 1011 0011_2

Binary Value:
0000 0000 0001 0000 1000 0000 1011 0011

To determine the instruction type, we decode the binary fields:

- opcode (7 bits): 0110011 → R-type instruction.
- rd (destination register, 5 bits): 00001 → register x1.
- funct3 (3 bits): 000 → indicates an addition operation.
- rs1 (source register 1, 5 bits): 00001 → register x1.
- rs2 (source register 2, 5 bits): 00001 → register x1.
- funct7 (7 bits): 0000000 → further specifies an addition.

Thus, this is an R-type instruction: add x1, x1, x1

7 b. Provide the instruction type, assembly language instruction, and binary representation of instruction described by the following RISC-V fields:

opcode=0x33, funct3=0x0, funct7=0x20, rs2=5, rs1=7, rd=6

You are given the following RISC-V fields:

- opcode: 0x33 (binary 0110011)
- funct3: 0x0 (binary 000)
- funct7: 0x20 (binary 0100000)
- rs2: 5 (x5)
- rs1: 7 (x7)
- rd: 6 (x6)

This is an R-type instruction. From the funct7 and funct3 fields, we know this is a subtraction operation (since funct7 = 0x20 indicates subtraction instead of addition).
Thus, the assembly instruction is: sub x6, x7, x5

Binary Representation: sub x6, x7, x5

- opcode: 0110011
- funct7: 0100000
- rs2: x5 (binary 00101)

- rs1: x7 (binary 00111)
- funct3: 000
- rd: x6 (binary 00110)

Putting all of these together, the binary representation is:
0100 0000 0101 0011 1000 0011 0011 0011
The hexadecimal representation of this is:
0x40538333

8. Your company has just bought a new Intel Core i5 dual core processor, and you have been tasked with optimizing your software for this processor. You will run two applications on this dual core, but the resource requirements are not equal. The first application requires 90% of the resources, and the other only 10% of the resources. Assume that when you parallelize a portion of the program, the speedup for that portion is 2. a. Given that 40% of the first application is parallelizable, how much speedup would you achieve with that application if run in isolation?

a. Given that 40% of the first application is parallelizable, how much speedup would you achieve with that application if run in isolation?

We use Amdahl's Law for speedup calculation when the application is run in isolation.

P (fraction parallelizable) = 0.4

Speedup of parallel portion = 2

The formula is:

Speedup = 1 / ((1 - P) + (P / speedup of parallel portion))

Speedup = 1 / ((1 - 0.4) + (0.4 / 2))

Speedup = 1 / (0.6 + 0.2)

Speedup = 1 / 0.8 = 1.25

Therefore, the speedup is 1.25.

b. Given that 99% of the second application is parallelizable, how much speedup would this application observe if run in isolation?

Similarly, using Amdahl's Law for the second application running in isolation:

- P (fraction parallelizable) = 0.99
- Speedup of parallel portion = 2

The formula is:
Speedup = 1 / ((1 - P) + (P / speedup of parallel portion))
Speedup = 1 / ((1 - 0.99) + (0.99 / 2))
Speedup = 1 / (0.01 + 0.495)
Speedup = 1 / 0.505 = 1.98

Therefore, the speedup is 1.98.

c. Given that 40% of the first application is parallelizable, how much overall system speedup would you observe if you parallelized it?

In this case, we consider the resource allocation of the system. The first application uses 90% of system resources, and of that 90%, 40% is parallelizable.

Resources used by the first application = 90%

Parallelizable portion of the system = 0.9 * 0.4 = 0.36 (36% of the total system resources)

Now applying Amdahl's Law:

Speedup = 1 / ((1 - P) + (P / S))

Where:

P = 0.36 (36% of the overall system is parallelizable)

S = 2 (parallelized portion runs with speedup of 2)

Calculation:

Speedup = 1 / ((1 - 0.36) + (0.36 / 2))

Speedup = 1 / (0.64 + 0.18)

Speedup = 1 / 0.82 = 1.2195 ≈ 1.22

Therefore, the overall system speedup is 1.22.

9. The Pentium 4 Prescott processor, released in 2004, had a clock rate of 3.6 GHz and voltage of 1.25 V. Assume that, on average, it consumed 10 W of static power and 90 W of dynamic power.

The Core i5 Ivy Bridge, released in 2012, has a clock rate of 3.4 GHz and voltage of 0.9 V. Assume that, on average, it consumed 30 W of static power and 40 W of dynamic power.

a. For each processor find the average capacitive loads.

$\frac{1}{2}CV^2F = \frac{1}{2}C(1.25)^2 \times 3.6$ GHz = 90 W [Prescott] => C = 32nF

$\frac{1}{2}CV^2F = \frac{1}{2}C(0.9)^2 \times 3.4$ GHz = 40 W [Ivy Bridge] => C = 29nF

b. Find the percentage of the total dissipated power comprised by static power and the ratio of static power to dynamic power for each technology

Prescott: Fraction of total power dissipation from Static = 10/100 = 10%

Static to Dynamic power ratio = 10:19 = 1:9 = 0.11

Ivy Bridge: Fraction of total power dissipation from Static = 30/70 = 42.9%

Static to Dynamic power ratio = 30:40 = 3:4 = 0.75

c. If the *total dissipated power is to be reduced by 10%, how much should the voltage be reduced* to maintain the same leakage current? Note: static power is defined as the product of voltage and current.

Total Power new / Total Power old = 0.9 (1a)

or, (Dn + Sn) / (Do + So) = 0.9 (1b)

Dynamic Power new = Dn = $C\,V_{new}^2 \times F$ (2)

Static Power new = Sn = $V_{new} \times I_L$ (3)

Static Power old = So = $V_{old} \times I_L$ (4)

For the Prescott:

From (2) V_{new} = sqrt (Dn/[C x F]) = $\sqrt{\frac{D_n}{32nF \times 3.6GHz}} = \sqrt{\frac{D_n}{115.2}}$ (5)

From (1b) Dn = 0.9 (So + Do) – Sn (6)

From (3), (4), Sn = So(V_{new}/V_{old}) (7)

Sn = V_{new} x (10W/1.25V) = V_{new} x 8 (8)

From (6), Dn = 0.9 x 100W – 8 V_{new}

So, Dn = [90-8V_{new}] (9)

From (5), (9) $V_{new} = \sqrt{\frac{90-8V_{new}}{32nF \times 3.6GHz}} =$

$$V_{new} = \sqrt{\frac{90-8V_{new}}{115.2}} \quad (10)$$

$$115.2V_{new}^2 = 90 - 8V_{new}$$

Solving above quadratic for V_{new}, we get

$V_{new} = 0.85V$ (11)

So, for the Prescott, operating voltage would scale down from 1.25V to 0.85V

10. A processor from Fasto Technologies (FT) has 32 registers, uses 16-bit immediates, and its ISA (Instruction Set Architecture) has 142 instructions. In a given program (assuming it has 100 instructions),

- 20 % of the instructions take 1 input register and have 1 output register.,
- 30 % have 2 input registers and 1 output register,
- 25 % have 1 input register, 1 output register and take an immediate input as well,
- The remaining 25 % have one immediate input and 1 output register.

(1) For each of the 4 types of instructions , how many bits are required? *Assume that the ISA requires that all instructions be a multiple of 8 bits in length*.

(2) How much less memory does the program take up if variable-length instruction set encoding is used as opposed to fixed-length encoding?

(1) 142 instructions ⇒ 8 bits (128 < 142 < 256); 32 registers ⇒ 5 bits; 16-bit immediates

- 1 reg in, 1 reg out: 8 + 5 + 5 = 18 bits ⇒ 24 bits
- 2 reg in, 1 reg out: 8 + 5 + 5 + 5 = 23 bits ⇒ 24 bits
- 1 reg in, 1 reg out, 1 imm: 8 + 5 + 5 + 16 = 34 bits ⇒ 40 bits
- 1 imm in, 1 reg out: 8 + 16 + 5 = 29 bits ⇒ 32 bits

(2) Since the largest instruction type requires 40-bit instructions, the fixed-length encoding will have 40 bits per instruction.

Thus, memory used by fixed-length encoding = 40*100 = 4000 bits

Whereas, each instruction type in the variable encoding will use the number of bits from 1:

$100*(0.2 * 24 + 0.3 * 24 + 0.25 * 40 + 0.25 * 32) = 3000$ bits,

ie 25 % less space.

11. Show how the value **0xedcba321** would be arranged in memory of a little-endian and a big-endian machine. Assume the data are stored starting at address 0 and that the word size is 4 bytes.

0xEDCBA321 split into words:

- 0xED (most significant byte)
- 0xCB
- 0xA3
- 0x21 (least significant byte)

Little Endian

The least significant byte (LSB) is stored at the lowest memory address, and the most significant byte (MSB) is stored at the highest memory address.

B3	B2	B1	B0
ed	cb	a3	21

Big Endian

The most significant byte (MSB) is stored at the lowest memory address, and the least significant byte (LSB) is stored at the highest memory address.

B3	B2	B1	B0
21	a3	cb	ed

12. Find the shortest sequence of RISC-V instructions that extracts bits 16 down to 11 from register x5 and uses the value of this field to replace bits 31 down to 26 in register x6 without changing the other bits of registers x5 or x6.

```
addi x7, x0, 0x3f   // Create bit mask for bits 16 to 11
slli x7, x7, 11     // Shift the masked bits
and x28, x5, x7     // Apply the mask to x5
slli x7, x7, 15     // Shift mask to cover bits 31 to 26
```

```
xori x7, x7,-1      // This is a NOT operation
and x6, x6, x7      // “Zero out” positions 31 to 26 of x6
slli x28, x28, 15   // Move selection from x5 into
                    // positions 31 to 26
or x6, x6, x28      // Load bits 31 to 26 from x28
```