Homework Set 1 - Hardware Architecture

Yiming Liu 2023010747

Problem 1

1. Note that only the change of sign bit might cause the overflow.

First, two positive number(take 0 into account) $A = 0A'$ and $B = 0B'$ adds up to C. (A', B' and C' above are shorthand for a unsigned 31-bit integer.)

If $C = 0C'$ is positive, $A + B = A' + B' = C' = C$. There is no overflow.

Else if $C = 1C'$ is negative, $A + B = A' + B' = 2^{31} + C' \geq 2^{31}$. Which means that $A + B$ is supposed to be $2^{31} + C'$, but the result is $C = -2^{31} + C' < 0$, which means that the overflow occurs.

From above analysis, we know that

$$A + B(\text{with truncation}) = \begin{cases} A + B - 2^{32} \text{ if } A + B \geq 2^{31} \\ A + B \text{ if } 0 \leq A + B < 2^{31} \end{cases}.$$

Next, when two negative number $A = 1A'$ and $B = 1B'$ adds up to C. Note that since we only preseve 32-bit signed integer, so we will need to truncate the result.

If after truncation, $C = 0C'$ is positive, $A + B = 10C'$ is supposed to be $-2^{32} + C' \in [-2^{32}, -2^{31} - 1)$. But the result is $C = C' > 0$, which means that the overflow occurs.

Else if after truncation $C = 1C'$ is negative, $A + B = 11C'$ is supposed to be $-2^{32} + 2^{31} + C' = -2^{31} + C' \in [-2^{31}, 0)$. And $C = -2^{31} + C'$ is the same as result, which means that there is no overflow.

From above analysis, we know that

$$A + B(\text{with truncation}) = \begin{cases} A + B + 2^{32} \text{ if } A + B < -2^{31} \\ A + B \text{ if} - 2^{31} \leq A + B < 0 \end{cases}.$$

What if A and B have different signs? Then $A + B \in [-2^{31}, 2^{31} - 1]$ so there is no overflow.

To sum up, we have the following formula:

$$A + B(\text{with truncation}) = \begin{cases} A + B - 2^{32} \text{ if } A + B \geq 2^{31} \\ A + B + 2^{32} \text{ if } A + B < -2^{31} \\ A + B \text{ if} - 2^{31} \leq A + B < 2^{31} \end{cases}.$$

And we know that iff A and B have the same sign, $A > 0, B > 0, A + B(\text{with truncation}) = A + B - 2^{32} < 0$ or $A < 0, B < 0, A + B(\text{with truncation}) = A + B + 2^{32} > 0$ will the overflow occurs .

2. (a) The error occurs when `i = 0`, the for loop will still do `i--` and check if `i > 0`. But `i` is unsigned int, so `i--` will cause `i` to be `2^32-1` and `i > 0` is still true, so the loop will continue.

The correct code should be:

```c
//Loop over an array in the inverse order
for (int i = 9; i >= 0; i--){
  printf("data %u = %d\n", i, a[i]);
}
```

(b) `lim` is a unsigned int, while `val` is a signed ine. So when the computer do the comparison, int will be cast to unsigned int. If `val` is negative(let's say -1), `check_data` should return 0. But during comparison, `val` will be cast to a large positive number(2^32 - 1), which is larger than `lim`, so the result is 1, which is different from what we want.

The correct code should be:

```c
// Check value range; output 1 if and only if value is > 10
constexpr int lim = 10;
int check_data(int val) {
  return val > lim ? 1 : 0;
}
```

(c) `buffer_size` is int while `data_size` is unsigned int. So when we do `buffer_size - data_size`, `buffer_size` will be cast to unsigned int. If `buffer_size` is negative, clearly `buffer_size - data_size` should be negative. But due to the casting, `buffer_size` will be a large positive number, so `buffer_size - data_size` may turn to be positive, which is not what we want.

The correct code should be:

```c
// Check if buffer has enough space to keep data
void copy_in(void* buffer, int buffer_size,
        void* data, int data_size) {
  if (buffer_size - data_size >= 0) memcpy(buf, data, data_size);
}
```

Problem 2

1. (a) `lw x2 112(x1)`

(b) `lw x1 12(x1)`

`lw x2 16(x1)`

2. To align the structure size on 64-bit machine, `char` and `int32_t` will have 7 bytes and 4 bytes padding respectively. Since `TreeNode*` use 8 bytes already, it will not have paddings. So, to access `val` starting from x3, we need to offset 24 bytes. Thus the answer is (d).

In the program, `TreeNode` doesn't use dynamic allocation, therefore we only consume on the stack.

For one `TreeNode`, we have to allocate $4 * 8 = 32$ bytes, so for `TreeNode arr[10]`, we need $32 * 10 = 320$ bytes.

Hence, we consume 320 bytes on the stack and 0 bytes on the heap.

3.

```c
# i in x1, A in x2, sum in x3
while(i != 0){
  sum += A[i];
  i--;
}
```

4. `bne` should change to `beq` and `addi x1, x1, 1` should change to `addi x1, x1, 4`. The correct code should be:

```c
    addi x4, x0, 0
L:  beq  x4, x2, E
    lw   x5, 0(x1)
    add  x3, x3, x5
    addi x4, x4, 1
    addi x1, x1, 4
    j    L
E:
```

Problem 3

1. We assume the total area is A, for a certain amount of area a, the number of instructions executed per second is $\eta\sqrt{a}$ where η is a constant.

We assume there are m instructions in total, mf of them are parallelizable, and $m(1-f)$ of them are serial.

Therefore, the time of a single big core is always $\frac{m}{\eta\sqrt{a}}$.

If we have N small cores with each has $\frac{a}{N}$ area, then we will need $\frac{m(1-f)}{\eta\sqrt{\frac{a}{N}}}$ time to execute the serial part and $\frac{mf}{N\eta\sqrt{\frac{a}{N}}}$ time to execute the parallel part.

Therefore, if we want multiple cores to perform better, we need to solve the following inequality:

$$\frac{m}{\eta\sqrt{a}} \geq \frac{m(1-f)}{\eta\sqrt{\frac{a}{N}}} + \frac{mf}{N\eta\sqrt{\frac{a}{N}}} = \frac{m(N-(N-1)f)}{N\eta\sqrt{\frac{a}{N}}}.$$

So $f \geq \frac{N-\sqrt{N}}{N-1}$.

Number of small cores N	Parallel portion f
2	$\left(2-\sqrt{2}\right) \approx 58.6\%$
4	$\frac{2}{3} = 66.7\%$
8	$\frac{8-2\sqrt{2}}{7} \approx 73.9\%$

2. Given that $f = 70\%$ and 8 small cores, the speedup is

$$\frac{\frac{m}{\eta\sqrt{a}}}{\frac{m(N-(N-1)f)}{N\eta\sqrt{\frac{a}{N}}}} = \frac{\sqrt{N}}{N-(N-1)f} \approx 0.91.$$

3. The hybrid design has a big core and 4 small cores, the time it will take is

$$\frac{mf}{4\eta\sqrt{\frac{a}{8}}} + \frac{m(1-f)}{\eta\sqrt{\frac{a}{2}}} = \frac{m(2-f)}{\eta\sqrt{2a}}$$

Therefore, the speedup it can achieve is

$$\frac{\frac{\frac{m}{\eta\sqrt{a}}}{m(2-f)}}{\eta\sqrt{2a}} = \frac{\sqrt{2}}{2-f} \approx 1.09(\text{compare to 1 big core})$$

$$\frac{\frac{\frac{m(N-(N-1)f)}{N\eta\sqrt{\frac{a}{N}}}}{m(2-f)}}{\eta\sqrt{2a}} = \frac{3.1}{2(2-f)} \approx 1.19(\text{compare to 8 small cores}).$$

Problem 4

1. We assume a B-byte block size and E-way set associative cache. Cache block size is C bytes. The total cache size is N bits. Memory address is m bits. And there are S sets.

For a memory address, we assume t bits are tag bits, s bits are set index bits, and b bits are block offset bits.

Thus, we have the following equations:

$$C = B \times E \times S$$
$$m = t + s + b$$
$$B = 2^b$$
$$S = 2^s$$
$$N = (1 + t + 8 \times B) \times E \times S$$

We already know C, m, B, E, so we can solve for S, t, s, N.

$$S = \frac{C}{BE}$$
$$b = \log_2(B)$$
$$s = \log_2(S) = \log_2\left(\frac{C}{BE}\right)$$

$$t = m - s - b = m - \log_2\left(\frac{C}{BE}\right) - \log_2(B) = m - \log_2\left(\frac{C}{E}\right)$$

$$N = \left(1 + m - \log_2\left(\frac{C}{E}\right) + 8B\right) \times E \times \frac{C}{BE} = \left(1 + m + 8B - \log_2\left(\frac{C}{E}\right)\right) \times \frac{C}{B}$$

Take $C = 32 \times 1024 = 2^{15}, m = 48, B = 64, E = 4$, we have $N = 274$ kbits.

Take $C = 32 \times 1024 = 2^{15}, m = 48, B = 8, E = 4$, we have $N = 400$ kbits.

Take $C = 32 \times 1024 = 2^{15}, m = 48, B = 64, E = 16$, we have $N = 275$ kbits.

2. For one level of cache, we know that $\text{AMAT} = \text{hit latency} + \text{miss rate} \times \text{miss penalty}$. $\text{miss penalty} = \text{time to access memory} + \text{time to deliver to cache} = \text{memory access latency}$.

(a) hit latency = 1 ns, hit rate = 60%, memory access latency = 100 ns, miss rate = 40%.

$\text{AMAT} = 1 + 0.4 \times 100 = 41$ ns.

(b) hit latency = 20 ns, hit rate = 90%, memory access latency = 100 ns, miss rate = 10%.

$\text{AMAT} = 20 + 0.1 \times 100 = 30$ ns.

(c) In this case, $\text{AMAT} = \text{first hit latency} + \text{first miss rate} \times \text{total miss penalty}$.

Since there are two level, total miss penalty $=$ second hit latency $+$ second miss rate' $\times$ miss penalty.

first hit latency = 1 ns, second hit latency = 20 ns, first miss rate = 40%, second miss rate' = $\frac{10\%}{40\%} = 25\%$, memory access latency = 100 ns.

$\text{AMAT} = 1 + 0.4 \times (20 + 0.25 \times 100) = 19 \text{ ns}.$

3. Let's say we have k instructions in total, then the execution time on an unrealistic perfect cache is k unit time.

If we execute on the real system, we denote m as the time of cache misses, then the execution time is $k + 100m$ unit time.

Therefore, slowdown = $\frac{k+100m}{k} = 1 + 100\frac{m}{k}$, which depends linearly on the ratio of its cache misses and instuction count.

We can also write slowdown = $1 + \frac{\text{MPKI}}{10}$. Which is more direct that miss rate.

4. (a) the cache is 512 bytes, direct-mapped, with 16-byte blocks size. Therefore, we have $S = \frac{512}{16} = 32$ sets.

The array is $x[2][128]$ has 256 elements, each element is 4 bytes, so the total size is $256 \times 4 = 1024$ bytes.

Therefore, the cache cannot hold the entire array, but only hold half of it.

When we access $x[1][i]$, it will be in the same set as $x[0][i]$, so the cache will have a conflict miss.

And we when access $x[0][i]$, it will have a cold miss.

So every hit will miss, the miss rates is 100%.

(b) If we double the cache capacity, the cache will be able to hold the entire array.

When we access $x[0][i]$ and $x[1][i]$, they will be in different sets, so there will be no conflict miss.

But when we access $x[0][4k]$ and $x[1][4k]$, we will have a cold miss.

After that, when we access $x[0][4k + j], x[1][4k + j]$ where $j = 1, 2, 3$, they will be cache hit since they are in the block of $x[0][4k]$ and $x[1][4k]$ respectively.

Therefore, the miss rates is 25%.

(c) If we use 2-way set associative cache, although the cache size is 512 bytes and can only hold half of the array, $x[0][i]$ and $x[1][i]$ will not conflict since one set can contain two blocks.

There will be conflict miss only between $x[0][4k], x[0][4k + 64]$ and between $x[1][4k], x[1][4k + 64]$ for $0 \leq k < 16$.

There will be cold miss when we access $x[0][4k]$ and $x[1][4k]$ for $0 \leq k < 16$.

So the miss rates is $\frac{64}{256} = 25\%$.

(d) No, larger cache size will not help. Since increasing cache size cannot reduce the cold miss, it may change conflict miss to cold miss, but the miss rate will not change.

(e) Yes, a larger block size will help. Since when we increase block size, after a cold miss, we will have more hits, which will reduce the miss rate.

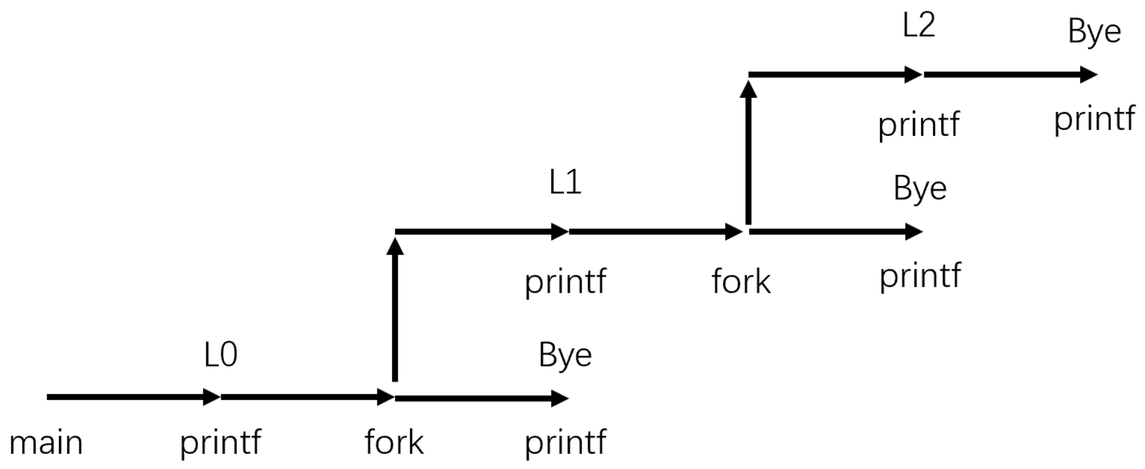
5. No, which is better depends on the what we want to access.

Take (c) cache as an example, if we want to access $x[0][0], x[1][0], x[0][64], x[0][0]$. For LRU replacement policy, we will have two cold miss and two conflict miss. The miss rate is 100%. But for random policy, 50% we will have two cold miss and two conflict miss, and 50% we will have two cold miss and one conflict miss. So the miss rate is 87.5%, which is lower than LRU miss rate.

Therefore, LRU is not always better than random policy.

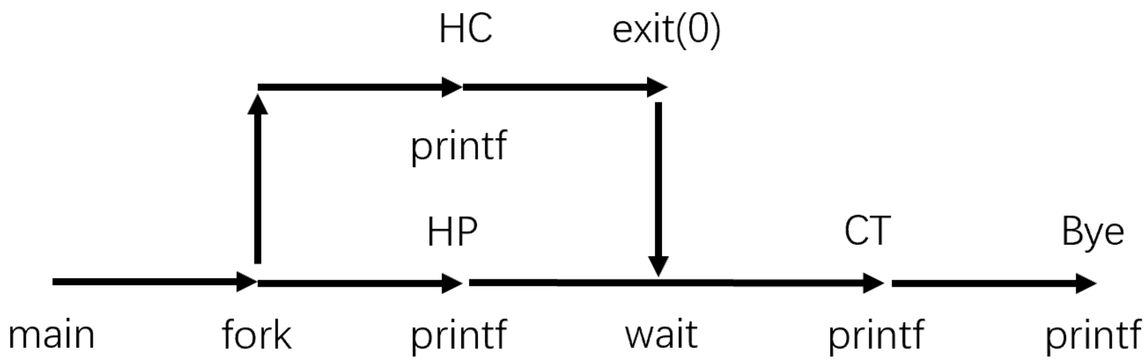
Problem 5

1. (a)



We can draw the process graph, and we know that only (ii) is possible.

(b)



We can draw the process graph, and we know that only (ii) is possible.

2. (a)

Policy	P_1	P_2	P_3	P_4	Average
FCFS	0	53	61	129	60.75
Worst FCFS	68	145	0	121	83.5
SJF	32	0	85	8	31.25
RR($q = 5$)	82	20	85	58	61.25

RR($q = 8$)	80	8	85	56	57.25
RR($q = 10$)	82	10	85	68	61.25
RR($q = 20$)	72	20	85	88	66.25

(b)

Policy	P_1	P_2	P_3	P_4	Average
RR($q = 5$)	136	30	141	90	99.25
RR($q = 8$)	112	10	119	74	78.75
RR($q = 10$)	110	12	115	86	80.75
RR($q = 20$)	86	22	101	100	77.25

Yes, the best time quantum choice change from $q = 8$ to $q = 20$.