**Problem 1: HCL (7 points)**

1. bool or = !(!a && !b);

2. Sol1: bool out = (v0 && v1) || (v1 && v2) || (v2 && v0);

Sol2:
```
bool out = [v0 && v1 && v2 : 1;
            v0 && v1       : 1;
            v1 && v2       : 1;
            v2 && v0       : 1;
            1              : 0;
           ];
```

*Or other solutions satisfied the truth table.

**Problem 2: Y86 (10 points)**

1. [1] irmovl Stack, %ebp [2] 30f154000000

[3] 2676 [4] 0x040

[5] jne Loop [6] 0x054

[7] efbe0000 [8] .pos 0x140

2. %eax = 0x0000beff (Sum of absolute value of four values)

**Problem 3 : Processor (18 points)**

1.

| Field | ssjxx |
|---|---|
| Fetch | icode:ifun ← $M_1$[PC]<br>rA:rB ← $M_1$[PC + 1]<br>valC ← $M_4$[PC + 2]<br>valP ← PC + 6 |
| Decode | valA ← R[rA] |
| Execute | Cnd ← Cond(CC, ifun) |
| Memory | -- |
| Write Back | -- |
| PC update | PC ← Cnd? valA : valC |

2. [1] E_icode == ISSJXX && e_Cnd

[2] -- [3] Bubble [4] Bubble

3. The origin design is better. The new design cannot take advantage of branch prediction. Because both address need to be read at DECODE period, it still need stall one cycle even though prediction is correct.

**Problem 4: Cache (32 points)**

1 [1] 27 [2] 3 [3] 2

2 64 bytes

3 [1] 0 [2] Miss [3] --
[4] 6 [5] Miss [6] --
[7] 1 [8] Hit [9] 0x22
[10] 0 [11] Miss [12] --
[13] 0.6

4 1) 3 * 4 * 16 = 192
2) (28 + 28 + 1) / 192 = 19/64
3) Both C1 and C2 can reduce the miss rate, the miss rate is 11/64.

**Problem 5: Memory Allocation (16 points)**

1.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16/1 | | | 16/1 | 8/0 | 8/0 | 24/1 | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 24/1 | 16/1 | | | 16/1 | 16/1 | | | 16/1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16/1 | | | 16/1 | 24/0 | | | | | 24/0 |

Internal: 41bytes

2.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16/1 | | | 16/1 | 8/0 | 8/0 | 16/1 | | | 16/1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 8/0 | 8/0 | 16/1 | | | 16/1 | 16/1 | | | 16/1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16/0 | | | 16/0 | 24/1 | | | | | 24/1 |

Internal: 41bytes

3. Let's count the read needed for each operation:

First-fit: 1+6+2+4+2+0=15

Best-fit:6+6+2+5+2+6=27

We can find that first-fit needs fewer reads to allocate the memory. Therefore, first-fit enjoys better performance.

**Problem 6: Optimization (17 points)**

1
```
int tmp1 = 0, tmp2 = 0, i;
// reduce loop overhead
int len = get_length(ra);
// reduce function call
state_result *sa = ra->data;
for (i = 0; i < len - 1; i += 2) {
  // two way loop unrolling + reassociation
  tmp1 = tmp1 + (sa[i].trump + sa[i].clinton);
  // two way loop unrolling + two accumulators
  tmp2 = tmp2 + (sa[i+1].trump + sa[i+1].clinton);
}
for (; i < len; i++)
  tmp1 += sa[i].trump + sa[i].clinton;
// reduce memory access
*total = tmp1 + tmp2;
```

2
```
int res = states[i].trump > states[i].clinton;
int winner = (res > 0) ? 0 : 1;
int high = (res > 0) ? states[i].trump : states[i].clinton;
int low = (res > 0) ? states[i].clinton : states[i].trump;
states[i].winner = winner;
states[i].gap = high - low;
```

3
```
// optimized code
int total_trump(state_result *r, int len) {
  if (len <= 0) return 0;
  return r[0].trump + r[1].trump + total_trump(r + 2, len - 2);
}

// invocation example
extern int length;
extern state_result *array;
int res;
if (length % 2 == 0)
  res = total_trump(array, length);
else
  res = total_trump(array + 1, length - 1);
```