Q1 Assembly programming

Please write the ARM assembly code corresponding to the following C code:

```
 int simple_calc (int x, int y)
{
   return (x - y);
}

int calculate (int a, int b, int c, int d)
{
   if (a > b)
   {
      return (c + d);
   }
   else
   {
      return simple_calc(c, d);
   }
}

void main()
{
   int a = 10, b = 20, c = 100, d = 200;
   int d = calculate(a, b, c, d);
}
```

Your assembly code should contain code and data sections. Please comment your code appropriately. **(Please translate the code as it is. That means do NOT apply any optimizations such as constant propagation, constant folding, dead code elimination etc.)**

Answer:

```
.section .text

simple_calc:
        sub r0, r2, r3          ; Do the subtraction
        mov r15, r14            ; Return to the caller

calculate:
        cmp r0, r1          ; Compare a and b
        ble else
        add r0, r2, r3      ; Add c and d
        b endif
  else:
        push {r14}          ; Save the return address register to stack as it will be overwritten on next call
        bl simple_calc
        pop {r14}           ; Pop the return address from stack
  endif:
        mov r15, r14        ; Return to the caller


main:
        ldr r4, =vars
        ldr r0, [r4]          ; load a
        ldr r1, [r4, #4]      ; load b
        ldr r2, [r4, #8]      ; load c
        ldr r3, [r4, #12]     ; load d
        bl calculate
        str r0, [r4, #12]     ;Save the returned value to d

.section .data

  vars:
        .word 10    ; a
        .word 20    ; b
        .word 100   ; c
        .word 200   ; d
```

Q2: DMA

Please explain what Direct Memory Access (DMA) is. What benefit does it provide? Where can the processor get data for calculations while DMA is in operation?

Answer:

DMA is a feature that uses a DMA controller to manage communication between I/O devices and main memory, thus freeing up the CPU. The CPU initializes bulk transfers by providing the DMA controller with the starting source and destination addresses, the number of bytes to transferred, and control messages like initiate transfer etc. The DMA controller sends an interrupt to the CPU when the data transfer is finished.

The benefit of DMA is that the CPU is free to perform other tasks when communication between memory and I/O devices is being controlled by DMA controller. During DMA, the CPU gets data from caches and internal registers. If the desired data is not available in caches or registers, the CPU must wait as the system bus is being controlled by DMA controller. Alternatively, DMA controller can periodically give bus control back to CPU so that it does not starve.

Q3: Processor Design

Please explain the difference between a single-cycle and pipelined processor design. What are the factors that can prevent a 3-stage pipelined processor from achieving 3x (three times) performance gain over single-cycle processor design?

Answer:

Instruction execution requires multiple steps like fetching the instruction, decoding, reading registers, and performing the desired operation. A single-cycle processor performs all these steps in one cycle to execute one instruction. Therefore, the cycle must be long enough to cover the combined latency of all these steps. This implementation results in low hardware utilization as hardware for only one of the steps is active at any time and the rest is idle. For example, when an instruction is being fetched, the hardware for decode and execution are idle.

In a pipelined processor, different steps required to execute an instruction are assigned to different pipeline stages. For example, one pipeline stage fetches instruction, another one decodes, and so on. As the amount of work to be done in one cycle is less, the clock cycle can be shorter and frequency can be higher. However, the key advantage of pipelined design is that it overlaps the execution of different parts of multiple instructions. For example, while one instruction is being fetched, another one can be decoded, and even a third one might be executing on functional units. This overlap improves hardware utilization and throughput.

Following factors prevent a three-stage pipeline from being 3x faster:

- Data and control dependencies

- Difficulties in dividing the instruction execution in three equal sized chunks which results in pipeline operating at a slightly higher frequency than 1/3 of the single cycle design.
- Pipeline filling and draining latency.

Q4: Caches

Consider a 4-way set-associative cache that uses 20-bits for tag, 6-bits for index, and 6-bits for byte offset. Find the cache size (capacity) in bytes. Please elaborate how you calculated the cache size.

Answer:

Cache size = number of sets * number of blocks/set * number of bytes/block

6-bit index means that there are 2^6 = 64 sets.
4-way set-associative means that there are 4 blocks/set.
6-bit byte offset means that there are 2^6 = 64 bytes/block.

Therefore, cache size is 64*4*64 = 16,384 bytes or 16KB.

Q5: Virtual Memory

Please elaborate how virtual memory solves capacity and safety issues in memory systems.

Answer:

Capacity: Virtual memory solves the capacity issue by introducing the notion of virtual and physical address spaces. Each application has access to full virtual address space (4GB for 32-bit address space) which gives the illusion that it owns the entire memory. However, the physical address space, which is the physical memory present in the system, might be much smaller than the virtual address space. To solve this capacity mismatch, some of the addresses from virtual space are mapped to physical memory and the rest are mapped to disk. When CPU accesses physical memory with a virtual address, the address is first translated to a physical address before accessing the memory. If the corresponding physical address is not in memory, it must be brought from disk to memory before serving the CPU request.

Safety: Virtual memory solves the safety issue by controlling what pages can be accessed by each application. It does so by means of permission bits on each page which control read, write and execute permissions to that page. In general, user space applications are not allowed access memory from OS/kernel space. Also, an application does not usually have access to memory pages of other applications. OS, in contrast, can access memory pages of applications.

Q6: Memory Regions

A program typically needs four memory regions for its execution: Stack, Heap, Static Data, and Instructions/Code. What is stored in each of these regions? Why do Stack and Heap grow in opposite directions (towards each other) rather than in the same direction?

Answer:

Stack: It is used to store local variables of a function, function parameters, return address, caller-save registers, and callee-save registers.

Heap: It is used to store dynamically allocated variables and is managed by programmers using *malloc* and *free* in C.

Static Data: It stores global and static variables.

Instruction/Code: It stores the instructions of the program.

Stack and heap grow towards each other to enable dynamic memory distribution between them based on application requirements. It ensures maximum memory utilization before the system runs out of memory.

Q7: C programming

In the code below, implement the update_bit function with three input arguments numA, numB, and pos. The function checks if the bit at position pos in number numA is set. If the bit is set, it toggles the bit at position pos in numB; otherwise it does not do anything. Finally, the function returns the value of numB.

Position is zero-based and starts from the right. The position of the least significant bit is 0.

For example, the bit on the position 3 in 9 (1001 in binary) is 1. Therefore, update_bit (9, 0, 3) should return 8 and update_bit (9, 0, 2) will return 0.

Please comment you code appropriately.

```
int update_bit (int numA, int numB, int pos)
{
   //Your code goes here
}
```

```
void main()
{
   printf("%d", update_bit (9, 0, 3));
}
```

Answer:

```
int update_bit (int numA, int numB, int pos)
{
   if (numA & (1 << pos)) {      //check if the bit at position “pos” is set in numA by anding it with 1.
      return (numB ^ (1 << pos)); // If the bit is set, flip the bit at “pos” in numB and return this value
   } else {
      return numB; //If the bit is not set, return the original numB
   }
}
```

Q8: Compilers

Apply these three optimization operations in order:

1. Dead code elimination
2. Constant propagation
3. Constant folding

to the following piece of C code repeatedly until no more optimizations are possible. Show the resulting code and indicate the modified and/or eliminated operations after each optimization step.

```
int q = 20;
int r = 11 - (q / 5);
int s;
s = r * 5;
if (s > 13) {
  s = r + 14;
}
return s * (100 / q);
```

Answers:

Pass1:

No dead code

Constant propagation:

```
int q = 20;
int r = 11 - (20 / 5);      //q replaced with 20
int s;
s = r * 5;
if (s > 13) {
  s = r + 14;
}
return s * (100 / 20);       //q replaced with 20
```

Constant folding:

```
int q = 20;
int r = 7;      //(11 - (20 / 5)) folded to 7
int s;
s = r * 5;
if (s > 13) {
  s = r + 14;
}
return s * 5;       //(100 / 20) folder to 5
```

Pass2:

Dead code elimination

```
~~int q = 20;~~         //Dead code as q has been propagated
int r = 7;
int s;
s = r * 5;
if (s > 13) {
  s = r + 14;
}
return s * 5;
```

Constant propagation:

```
int r = 7;
int s;
s = 7 * 5;                    //r replaced with 7
if (s > 13) {
  s = 7 + 14;                 //r replaced with 7
```

```
}
return s * 5;
```

Constant folding:

```
int r = 7;
int s;
s = 35;              //(7 * 5) folded to 35
if (s > 13) {
  s = 21;            //(7 + 14) folded to 21
}
return s * 5;
```

Pass3:

Dead code elimination

```
~~int r = 7;~~      //Dead code as r has been propagated
int s;
s = 35;
if (s > 13) {
  s = 21;
}
return s * 5;
```

Constant propagation:

```
int s;
s = 35;
if (35 > 13) {      //s replaced with 35
  s = 21;
}
return s * 5;
```

Constant folding:

```
int s;
s = 35;
if (true) {          //(35 > 13) folded to true
  s = 21;
}
return s * 5;
```

Pass4:

Dead code elimination

```
int s;
s = 35;
if (35 > 13) {      //Dead code as the condition is always true
  s = 21;
}
return s * 5;
```

Constant propagation:

```
int s;
s = 35;
s = 21;

return 21 * 5;       //s replaced with 21
```

Constant folding:

```
int s;
s = 35;
s = 21;

return 105;       //(21 * 5) folded to 105
```

Pass5:

Dead code elimination

```
int s;
s = 35;
s = 21;

return 105;
```

No constant propagation and no constant folding

Final code:

```
return 105;
```

Q9: Operating Systems

The most simple operating system process model consists of the three process states running, ready, and blocked and related transitions between these states.

Describe all the transitions between these process states that take place when:

1. A process requests a long-running I/O operation from the operating system
2. A long running I/O operation completes and notifies the operating system of its completion
3. A timer interrupt occurs

For each transition, indicate in a few words why this transition occurs.
If there are multiple different options for transitions that can occur, please describe all of them.

Answer:

1. On requesting a long-running I/O, the process goes from **running** state to **blocked** state because it cannot execute until the requested I/O has completed.
2. When a long running I/O completes, the process goes from **blocked** state to **ready** state because the I/O has now finished and the process can restart execution. The process will be picked for execution based on the scheduling policy.
3. The process goes from **running** state to **ready** state when timer interrupt occurs as it has timed-out. (There are other possibilities, but I suppose this answer is sufficient.)

Q10: Power consumption

Please explain how Clock Gating and Power Gating work. What type of power consumption do they reduce?

Answer:

Clock gating is a technique to temporarily cut-off the clock signal from a given circuit. A simple implementation involves passing the clock signal through an AND gate before connecting it to the desired circuit. The other input of the AND gate controls whether the clock reaches the circuit. When a circuit is not in use, the clock can be disabled to reduce its **dynamic power consumption**. As dynamic power consumption is directly proportional to clock frequency, making the frequency zero eliminates dynamic power consumption as long as the circuit is clock gated.

Power gating is a technique to temporarily disconnect the power supply from a circuit. A simple implementation includes a sleep transistor (usually in a higher technology node that the circuit itself) between the power supply (or ground) node and the circuit itself. By switching the sleep transistor on or off, the circuit can be connected or disconnect to/from the power supply. As power gating switches off the power supply to the circuit, **it eliminates both static and dynamic power consumptions**.