Department of Computer Science & Engineering

University of Dhaka

CSE 2106: Microprocessor and Assembly Lab

Assignment

Assignment-1: SmartCare-32 Assembly Project REPORT

Submission Date: 15 December 2025

Submitted by-

1.Farhan Labib Ahan	Roll- 08
2.Dipa Biswas	Roll- 51
3.Shahriar Islam	Roll -61

SmartCare-32 Assembly Project REPORT

Module 1: Patient Record Initialization (module_1.s)-

1. Introduction

Module 1 is responsible for initializing patient records in the SmartCare-32 system. Whenever a patient is admitted, their personal and medical information must be stored in memory in a structured and organized format. This module prepares patient data in predefined RAM locations so that subsequent modules (billing, treatment, pharmacy, etc.) can reliably access and process this information.

The implementation is written in ARM assembly language and focuses on correct memory layout, alignment, and initialization of multiple patient records.

2. Objective of the Module

The primary objectives of Module 1 are:

- To define a **patient data structure** with fixed offsets for each field
- To initialize **three patient records** with test data
- To store each record at a **predefined RAM address**
- To ensure **correct alignment** for byte, halfword, and word data types
- To provide functions that return the base address of each patient record

3. Patient Data Structure Design

Each patient record occupies **32 bytes** in RAM. The structure layout is carefully designed to meet alignment requirements of the ARM architecture.**Memory Layout:**

Offset (Base)	Field Name	Size	Description
0	Patient ID	4 bytes	Unique 32-bit identifier
4	Name Pointer	4 bytes	Address of patient name string
8	Age	1 byte	Patient age in years
10	Ward Number	2 bytes	Hospital ward number
12	Treatment Code	1 byte	Encoded treatment type
16	Room Daily Rate	4 bytes	Cost per day
20	Medicine List Pointer	4 bytes	Address of medicine list

Unused bytes act as padding to preserve alignment.

4. Memory Allocation Strategy

Each patient record is stored in a fixed RAM location:

Patient	RAM Address
Patient 1	0x20000000
Patient 2	0x20000020
Patient 3	0x20000040

The 32-byte spacing ensures that each patient structure does not overlap and remains word-aligned.

5. Patient Initialization Details

Patient 1: Shahriar Samrat

- **Patient ID:** 0x00001001
- **Age:** 35
- **Ward:** 201
- **Treatment Code:** 1
- **Room Rate:** 2500
- **Medicine List:** med_list1

Patient 2: Dipa Biswas

- **Patient ID:** 0x00001002
- **Age:** 28
- **Ward:** 305
- **Treatment Code:** 2
- **Room Rate:** 1800
- **Medicine List:** med_list2

Patient 3: Farhan Labib

- **Patient ID:** 0x00001003
- **Age:** 42
- **Ward:** 102
- **Treatment Code:** 3
- **Room Rate:** 3200
- **Medicine List:** med_list3

All name strings, IDs, and medicine lists are imported from a separate data file (data.s), ensuring modularity and separation of data from logic.

6. Functional Overview

6.1 init_patients

This function initializes all patient records by sequentially calling:

- **init_patient1**
- **init_patient2**
- **init_patient3**

It preserves the link register and ensures safe function return.

6.2 Individual Patient Initialization Functions

Each init_patientX function:

1. Loads the base address of the patient structure
2. Stores the patient ID using word access (STR)
3. Stores the name pointer
4. Stores age using byte access (STRB)
5. Stores ward number using halfword access (STRH)
6. Stores treatment code using byte access
7. Stores room rate as a word
8. Stores the medicine list pointer

This demonstrates correct use of ARM load/store instructions for different data sizes.

6.3 Getter Functions

The following functions return the base address of each patient record:

- get_patient1
- get_patient2
- get_patient3

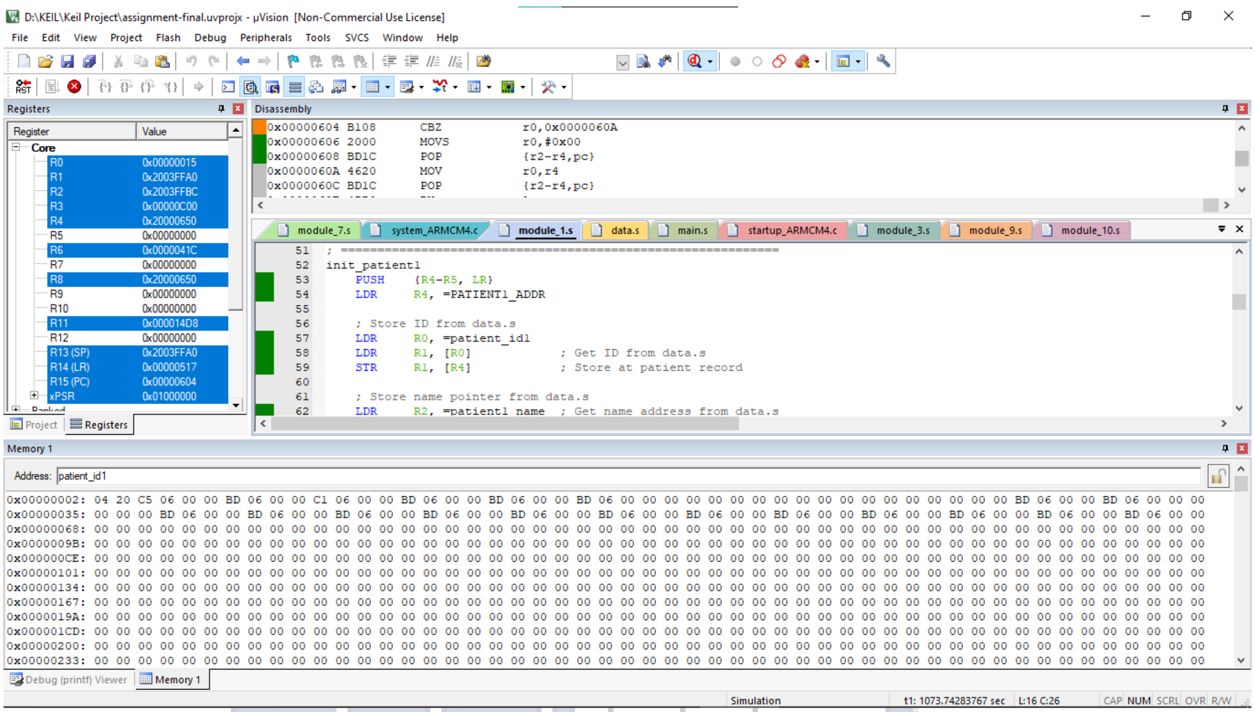
These functions allow other modules to access patient data without hardcoding addresses.

7. Alignment and Data Integrity

- **Word-aligned fields** (ID, pointers, room rate) are stored at offsets divisible by 4
- **Halfword fields** (ward number) are stored at even addresses
- **Byte fields** (age, treatment code) use STRB
- Padding ensures no misaligned memory access occurs

This guarantees compatibility with ARM memory access rules and prevents runtime faults.

8.Output Screenshot:



Module 2: Vital Sign Data Acquisition (module_2.s)-

1. Introduction

Module 2 performs **vital sign monitoring** for admitted patients. In the SmartCare-32 scenario, each patient is connected to sensor devices that continuously produce vital values. Since real sensors are not available in this simulation, the sensor outputs are represented using **fixed memory-mapped addresses** in RAM. The module reads these values and stores them into rolling history buffers for later analysis (alerts, averaging, trending, etc.).

2. Objective of the Module

This module is designed to:

- Read **Heart Rate (HR)**, **Blood Pressure (BP)**, and **Oxygen Saturation (O_2)** values from predefined memory addresses.
- Maintain a **rolling buffer of 10 readings** for each vital sign.
- Implement **circular buffer rotation** using modulo arithmetic (wraparound at index 10).
- Use correct ARM **load/store instructions**, pointer addressing, and index increment logic.
- Support **three patients**, each having three vital signs → **9 buffers total**.

3. Sensor Memory Map (Simulated)

Each patient has three sensor registers stored in fixed RAM locations:

Patient	HR Address	BP Address	O_2 Address
1	0x20001000	0x20001004	0x20001008
2	0x2000100C	0x20001010	0x20001014
3	0x20001018	0x2000101C	0x20001020

The module reads each sensor value as a **32-bit word** using LDR.

4. Buffer Design and Data Structures

4.1 Buffers

There are **9 buffers** total:

- **HR buffers:** hr1_buffer, hr2_buffer, hr3_buffer
- **BP buffers:** bp1_buffer, bp2_buffer, bp3_buffer
- **O_2 buffers:** o21_buffer, o22_buffer, o23_buffer

4.2 Buffer Size

Each buffer stores:

- **10 readings**
- Each reading = **4 bytes**
- Total per buffer = **40 bytes**

4.3 Index Tracking

Each buffer has its own index variable:

- **HR indices:** hr1_index, hr2_index, hr3_index
- **BP indices:** bp1_index, bp2_index, bp3_index
- **O_2 indices:** o21_index, o22_index, o23_index

Each index ranges from **0 to 9**, and is updated as a circular counter.

5. Program Flow and Functional Overview

5.1 Entry Point: module_two

The function module_two serves as the module entry. It performs two major tasks:

1. **Reset all buffer indices to 0**
    - Ensures clean start and predictable buffer positions.
2. **Read all patient vitals**
    - Calls read_all_patients to gather and store one set of readings per patient.

The function terminates cleanly using BX LR.

6. Vital Reading Sequence

6.1 read_all_patients

This function sequentially reads vitals from:

- read_patient1
- read_patient2
- read_patient3

It uses PUSH {LR} and POP {PC} to preserve return flow.

6.2 Per-Patient Reading

Each patient reading function follows the same 3-step pattern:

1. Read HR value from HR address → store in HR buffer
2. Read BP value from BP address → store in BP buffer
3. Read O_2 value from O_2 address → store in O_2 buffer

For every vital sign, it calls: store_reading(reading_value, buffer_addr, index_addr)

7. Core Algorithm: Circular Buffer Storage

7.1 Function: store_reading

Inputs

- R1 = current sensor reading value
- R2 = base address of buffer
- R3 = address of index variable

Steps Implemented

1. Load current index from [R3] into R4
2. Compute write location using scaled addressing:
    - buffer + (index × 4)
3. Store the value into the buffer using:
    - STR R1, [R2, R4, LSL #2]
4. Increment index:
    - index = index + 1
5. Wraparound using modulo logic:
    - if index == 10 → reset index to 0
6. Store updated index back into [R3]

This logic creates a **rolling history**, meaning after 10 readings the buffer begins overwriting the oldest readings (circular queue behavior).

8. ARM Instruction Usage and Requirements Compliance

Technical Requirement Check

Read simulated vital values from fixed memory

- Done using LDR R1, [R0] after loading sensor address into R0

Maintain rolling buffer of 10 entries per vital

- 40-byte buffers and indices ensure 10 entries are stored

Implement buffer rotation using modulo arithmetic

- CMP R4, #10 + MOVGE R4, #0 implements wraparound

Use LD/ST and pointer increment logic

- Uses LDR, STR, and index-based addressing (LSL #2)

Supports all 3 patients

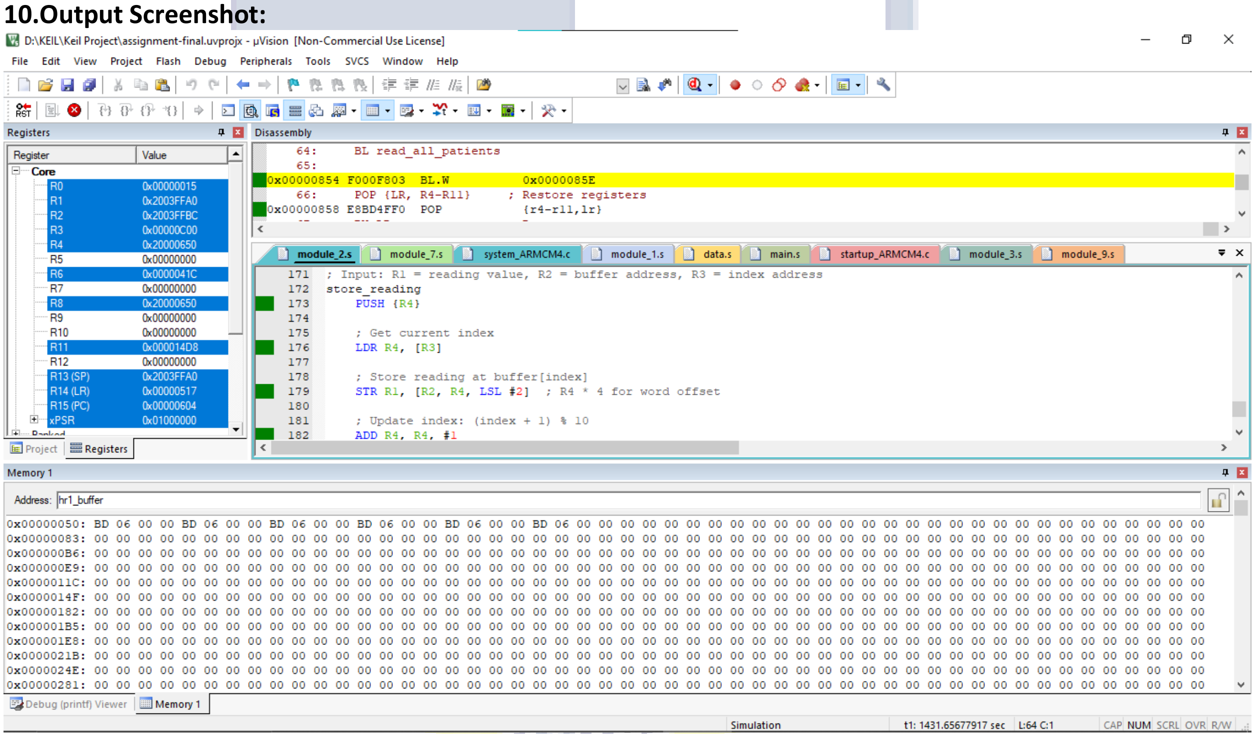
- Separate functions and buffers for Patient 1, 2, and 3

9. Observations

- The module currently reads **one set of vitals per execution**. In a real monitoring system, module_two would typically be called repeatedly inside a loop or timer interrupt to continuously log new readings.

- Each vital sign is stored as a **32-bit word**, which is simple and safe for alignment, even though real HR/O_2 values could fit in smaller sizes.

10.Output Screenshot:



Module 3: Vital Threshold Alert Module (module_3.s)-

1. Introduction

Module 3 is responsible for **alert management** in SmartCare-32. In the hospital workflow, nurses must be warned immediately when a patient's vital signs cross dangerous clinical thresholds. This module simulates that behavior by checking new vital readings against preset limits and updating alert counters (and, indicates how alert records should be stored in memory for later review).

2. Objective of the Module

This module is designed to:

- Compare incoming readings with medical thresholds:
  - **HR > 120** → High Heart Rate Alert
  - **O_2 < 92** → Low Oxygen Alert
  - **SBP > 160 or SBP < 90** → Abnormal Blood Pressure Alert
- On threshold violation:
  - Set a **1-byte alert flag**
  - Create an **alert record** (vital type, actual reading, timestamp)
  - Insert the record into an **alert buffer** in RAM
- Maintain per-patient alert tracking:
  - patient_alert_count1, patient_alert_count2, patient_alert_count3

3. Medical Thresholds Used

Vital Sign	Condition	Threshold
Heart Rate	High	HR > 120
Oxygen Saturation	Low	$O_2 < 92$
Systolic BP	High	SBP > 160
Systolic BP	Low	SBP < 90

4. Alert Record Format (Memory Layout)

The specification requires an alert record stored in RAM whenever an abnormal vital occurs.

Required Alert Record Fields

A complete alert record should contain:

- **Vital type** (HR / BP / O_2)

- **Actual reading**
- **Timestamp** (counter-based)
- **Padding/alignment** for clean word access

A clean 16-byte design (matches the requirement) is typically:

Offset	Size	Field
+0	1 byte	Vital type
+1	1 byte	Severity (optional but useful)
+2	2 bytes	Padding
+4	4 bytes	Actual reading
+8	4 bytes	Timestamp
+12	4 bytes	Reserved / future use

Note: Your written “Alert Memory Layout” earlier shows only type/severity/padding/timestamp. The technical requirement also needs **actual reading**, so it must be included in the 16-byte record.

5. Test Scenario Implemented

The module uses fixed test values to force known alerts:

Patient 1

- HR = 130 (**HIGH**)
- BP = 170 (**HIGH**)

- O_2 = 95 (Normal)
  Total: **2 alerts**

Patient 2

- HR = 110 (Normal)
- BP = 120 (Normal)
- O_2 = 85 (**LOW**)
  Total: **1 alert**

Patient 3

- HR = 130 (**HIGH**)
- BP = 170 (**HIGH**)
- O_2 = 85 (**LOW**)
  Total: **3 alerts**

Final expected counts:

- patient_alert_count1 = 2
- patient_alert_count2 = 1
- patient_alert_count3 = 3

6. Program Flow and Functional Overview

6.1 Entry Point: module_three

module_three initializes the alert system state:

- Resets alert_index1, alert_index2, alert_index3
- Resets per-patient alert counters to 0
- Resets timestamp_counter to 0
- Runs three test cases by calling check_patient_vitals for each patient

It ends with BX LR and returns to the caller.

7. Vital Checking Logic

7.1 check_patient_vitals

Inputs:

- R0 = patient number (1–3)
- R1 = HR
- R2 = BP
- R3 = O_2

It calls three subroutines:

- check_hr_for_patient
- check_bp_for_patient
- check_o2_for_patient

Each routine performs threshold comparisons and calls increment_alert_count if abnormal.

8. Alert Counting Mechanism

8.1 increment_alert_count

This function updates the correct patient counter:

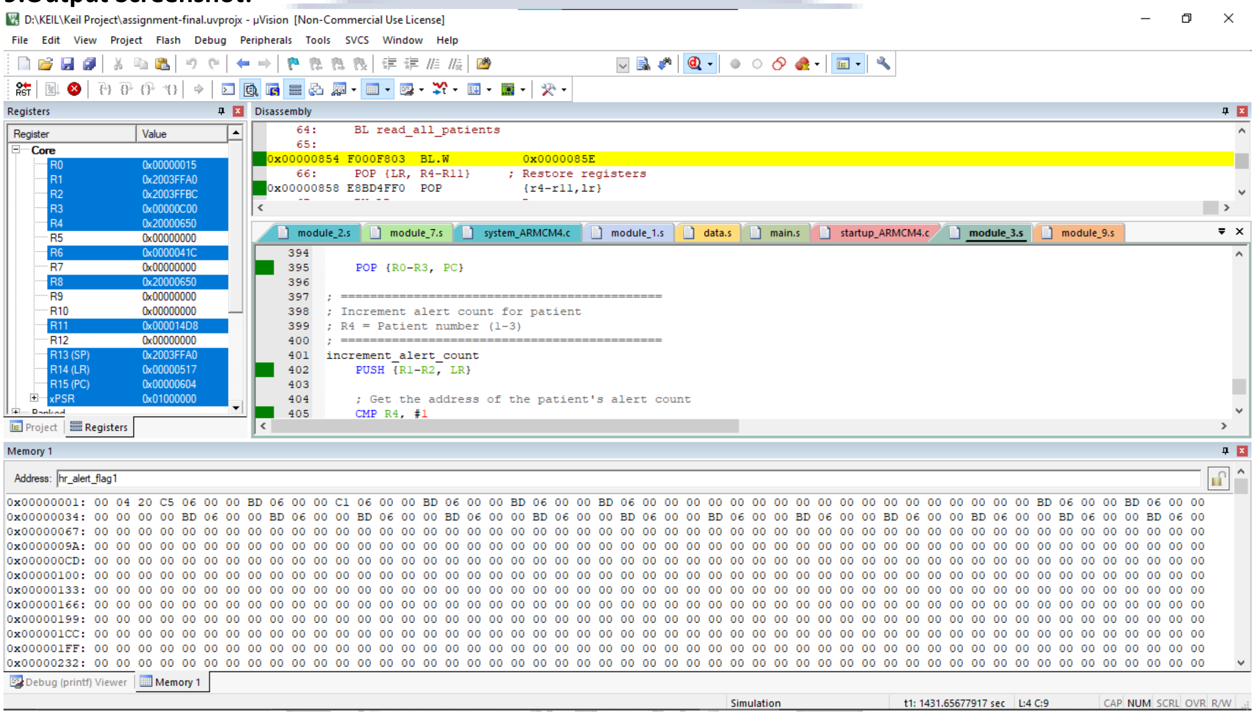
- If R0 == 1 → patient_alert_count1
- If R0 == 2 → patient_alert_count2
- Else → patient_alert_count3

Then it:

1. Loads the current count
2. Increments by 1
3. Stores the updated count back into memory

This produces correct totals for the test scenario.

9.Output Screenshot:



Module 4: Medicine Administration Scheduler (module_4.s)-

1. Objective

Maintain a medication schedule for **three medicines (MED1, MED2, MED3)** using a global time counter. The module updates the internal clock, calculates the next due time for each medicine, and sets a **due / not-due flag** by comparing it with the current clock value.
If a medicine is due, the **last administered time is updated**.

2. Data Used (RAM Variables)

Global Clock

• CLOCK_COUNTER : Global time counter (increments each scheduler run)

Medicine 1

• MED1_LAST : Last time MED1 was administered
• MED1_INTERVAL : Prescribed interval between doses
• MED1_NEXT : Computed next due time (output)
• MED1_FLAG : Dose due flag (output)

Medicine 2

• MED2_LAST : Last time MED2 was administered
• MED2_INTERVAL : Prescribed interval between doses
• MED2_NEXT : Computed next due time (output)
• MED2_FLAG : Dose due flag (output)

Medicine 3

• MED3_LAST : Last time MED3 was administered
• MED3_INTERVAL : Prescribed interval between doses
• MED3_NEXT : Computed next due time (output)
• MED3_FLAG : Dose due flag (output)

3. Solution Technique (logic pattern)

Update Clock Logic (update_clock)

1. Load CLOCK_COUNTER
2. Add 1 tick (hour)
3. Store updated value back into CLOCK_COUNTER

Dosage Scheduling Logic (check_medicine_1 / check_medicine_2 / check_medicine_3)

For each medicine (MED1, MED2, MED3):

1. Load CLOCK_COUNTER (current time)
2. Load MEDx_LAST and MEDx_INTERVAL
3. Compute next due time:
   MEDx_NEXT = MEDx_LAST + MEDx_INTERVAL
4. Compare CLOCK_COUNTER with MEDx_NEXT
5. If CLOCK_COUNTER >= MEDx_NEXT:
   - Set MEDx_FLAG = 1 (Dose due)
   - Update MEDx_LAST = CLOCK_COUNTER
6. Else:
   - Set MEDx_FLAG = 0 (Not due)

4. Function Call Structure

This module **does not use register-passed parameters**.
All routines directly load required variables from memory.

Execution order inside module_four:

1. Call update_clock
2. Call check_medicine_1
3. Call check_medicine_2
4. Call check_medicine_3

This is done in module_4.s.

5. Step-by-Step Scheduling Logic (As Executed)

Step 1: Update Clock

• CLOCK_COUNTER = CLOCK_COUNTER + 1

Step 2: Check a Medicine (Example: MED1)

1. Load last time
   o r1 = [MED1_LAST]
2. Load interval
   o r2 = [MED1_INTERVAL]
3. Compute next due
   o r3 = r1 + r2
4. Store next due
   o [MED1_NEXT] = r3
5. Load current time
   o r0 = [CLOCK_COUNTER]

6. Compare
   o If r0 < r3 → NOT_DUE
   o Else → DUE
7. Set flag and update last time
   o Due →
   • [MED1_FLAG] = 1
   • [MED1_LAST] = r0
   o Not due →
   • [MED1_FLAG] = 0

The same logic applies for **MED2 and MED3**.

6. Module Output (What changes in memory)

After update_clock runs:

• CLOCK_COUNTER = CLOCK_COUNTER + 1

For each medicine MEDx:

• MEDx_NEXT = MEDx_LAST + MEDx_INTERVAL

If dose is NOT due:

• MEDx_FLAG = 0
• MEDx_LAST unchanged

If dose IS due:

• MEDx_FLAG = 1
• MEDx_LAST = CLOCK_COUNTER

7. Output Example Scenarios (Example: MED1)

Scenario 1: Not Due

Assume:
• MED1_LAST = 4
• MED1_INTERVAL = 4
• CLOCK_COUNTER = 5

Then:
• MED1_NEXT = 8
• 5 < 8 ⇒ MED1_FLAG = 0

Output:
• MED1_NEXT = 8
• MED1_FLAG = 0
• MED1_LAST = 4

Scenario 2: Due

Assume:

- MED1_LAST = 6
- MED1_INTERVAL = 4
- CLOCK_COUNTER = 10

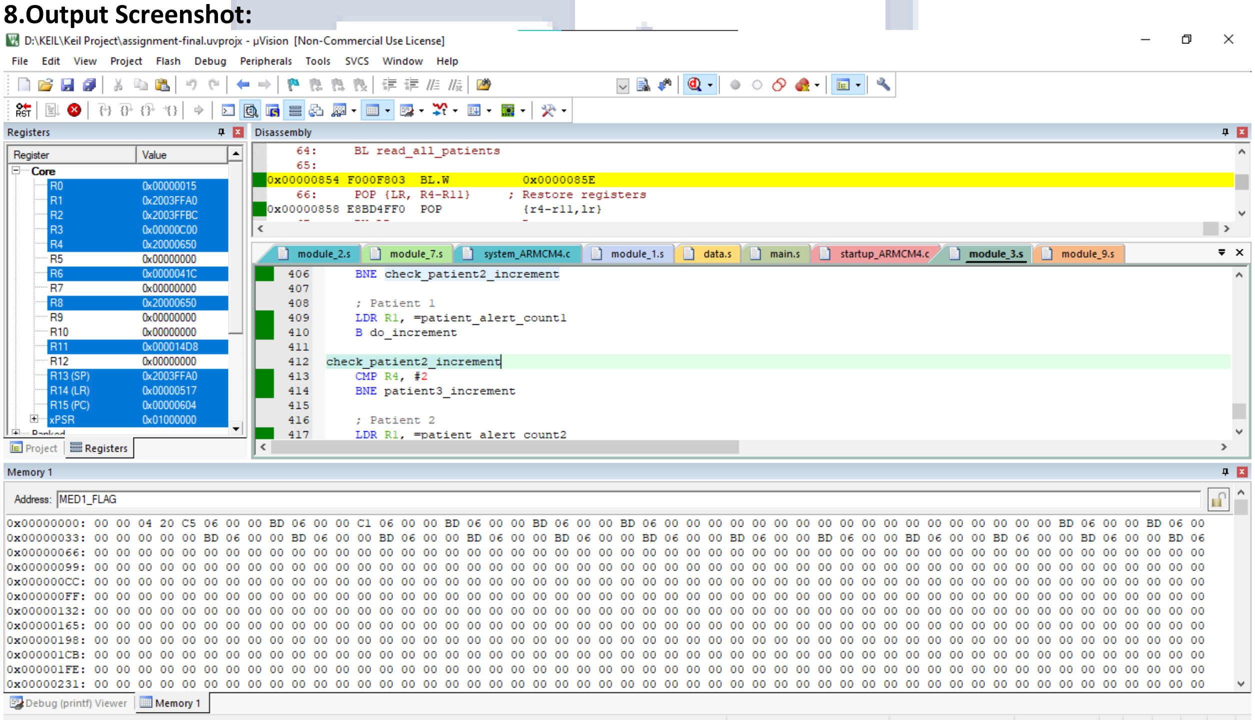
Then:

- MED1_NEXT = 10
- 10 ≥ 10 ⇒ MED1_FLAG = 1

Output:

- MED1_NEXT = 10
- MED1_FLAG = 1
- MED1_LAST = 10

8.Output Screenshot:



MODULE 5: Treatment Cost Computation (module_5.s):

1. Objective

Given a **treatment code (0–99)**, this module looks up the corresponding **treatment cost** from a table and stores the result in memory. If the code is invalid (>99), it stores **0** as the cost.

2. Data Used

Input

- **TREATMENT_CODE :** holds the treatment code number (expected range **0–99**)

Output

- **TREATMENT_COST** : stores the cost retrieved from the table (or **0** if invalid)

3. Lookup Table

- **TREATMENT_TABLE** : 100 entries (index **0 to 99**)
    - Entry format: **word (4 bytes)** per cost
    - Values are: **100, 200, 300, ... , 10000** (increments of 100)

4. Solution Technique

1. Load address of input variable **TREATMENT_CODE** into R0
2. Load address of output variable **TREATMENT_COST** into R1
3. Call **Compute_Treatment_Cost**
4. Stop in an infinite loop for memory inspection

5. Cost Computation (Compute_Treatment_Cost)

1. Load **code = [TREATMENT_CODE]**
2. Validate code:
    - If **code > 99 → store 0 to TREATMENT_COST**
3. If valid:
    - Compute table offset: **offset = code * 4**
    - Compute entry address: **&TREATMENT_TABLE + offset**
    - Load cost from table
    - Store cost into **TREATMENT_COST**

This uses **table indexing** in ARM assembly (base address + scaled index).

6. Inputs Passed (Register Passing)

Before calling **Compute_Treatment_Cost:**

- **R0 = &TREATMENT_CODE**
- **R1 = &TREATMENT_COST**

7. Step-by-step logic

1. **Load treatment code**
    - **r4 = [r0]** → r4 = code
    - CMP r4, #99
    - If **r4 > 99** → branch to invalid_code
2. **Load table base**
    - **r5 = &TREATMENT_TABLE**
3. **Compute offset**
    - **r4 = r4 << 2** → multiply by 4 (word size)
4. **Compute address of entry**
    - **r6 = r5 + r4**
5. **Read cost**
    - **r7 = [r6]**
6. **Store output**
    - **[r1] = r7** → writes into TREATMENT_COST
7. **Invalid code case**
    - **r7 = 0**
    - **[r1] = 0**

8. Module Output (in memory)

After Compute_Treatment_Cost runs:

- If **0 ≤ TREATMENT_CODE ≤ 99**
  TREATMENT_COST = TREATMENT_TABLE[TREATMENT_CODE]
- If **TREATMENT_CODE > 99**
  TREATMENT_COST = 0

9. Output (Examples)

Example 1: Valid code

Assume:

- **TREATMENT_CODE = 0**

Then:

- **TREATMENT_COST = TREATMENT_TABLE[0] = 100**

Output: **TREATMENT_COST = 100**

Example 2: Valid code

Assume:

- **TREATMENT_CODE = 5**

Then:

- **TREATMENT_COST = TREATMENT_TABLE[5] = 600**

Output: **TREATMENT_COST = 600**

Example 3: Valid code (last entry)

Assume:

- **TREATMENT_CODE = 99**

Then:

- **TREATMENT_COST = TREATMENT_TABLE[99] = 10000**

Output:**TREATMENT_COST = 10000**

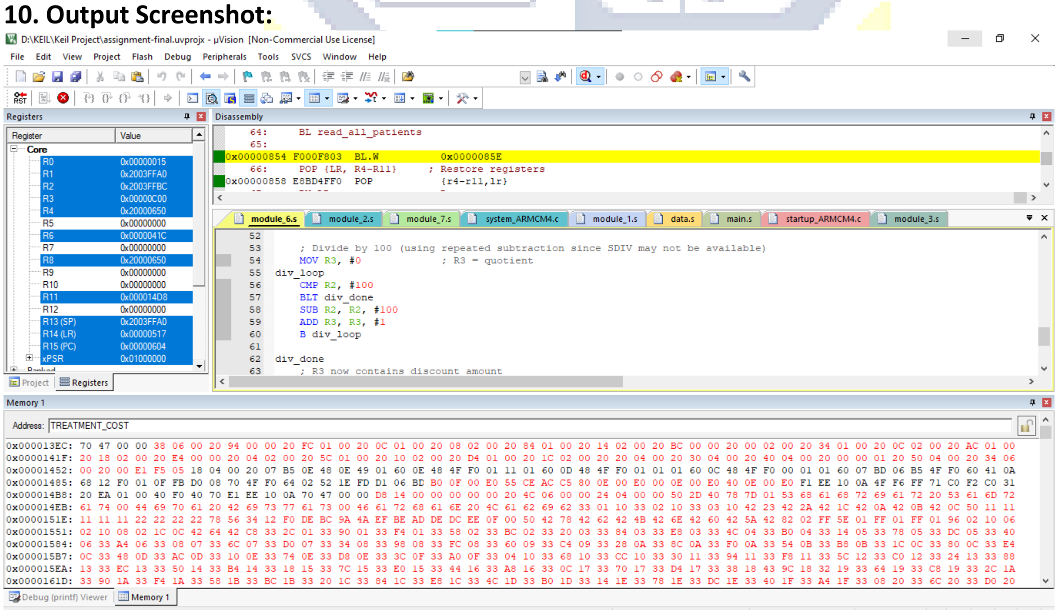
Example 4: Invalid code

Assume:

- **TREATMENT_CODE = 120**
- **120 > 99 ⇒ invalid**

Output:TREATMENT_COST = 0

10. Output Screenshot:



MODULE 6: Daily Room Rent Calculation(module_6.s)

1.Objective

Compute a patient's **total room rent cost** using:

- **ROOM_COST = ROOM_RATE × DAYS_STAYED**
- **If DAYS_STAYED > 10, apply a 5% discount to the total**

Final cost is written back to memory in **ROOM_COST**.

2. Data Used

Input

- **ROOM_RATE :** daily room charge (per day)
- **DAYS_STAYED :** number of days stayed

Output

- **ROOM_COST :** total final amount after discount (if any)

3. Solution Technique

1. Load addresses of input/output variables into registers:
   - R0 = &ROOM_RATE
   - R1 = &DAYS_STAYED
   - R2 = &ROOM_COST
2. Call Calc_Room_Rent
3. Stop in an infinite loop to inspect memory

4. Cost Computation (Calc_Room_Rent)

1. Load inputs:
   - **rate = [ROOM_RATE]**
   - **days = [DAYS_STAYED]**
2. Compute base cost:
   - **cost = rate × days**
   - Store into **[ROOM_COST]**
3. Discount check:
   - If **days <= 10** → **no discount**, return
   - If **days > 10** → compute discount and subtract
4. Discount formula:
   - **discount = (cost × 5) / 100**
   - **final_cost = cost - discount**
   - Store final result into **[ROOM_COST]**

This uses **integer arithmetic** (no floating point), so discount is truncated if not divisible evenly.

5. Inputs Passed

Before calling **Calc_Room_Rent:**

- **R0 = &ROOM_RATE**
- **R1 = &DAYS_STAYED**
- **R2 = &ROOM_COST**

6. Step-by-step logic

1. **Load room rate**
   - **r4 = [r0] → r4 = rate**
2. **Load days stayed**
   - **r5 = [r1] → r5 = days**
3. **Compute initial room cost**
   - **r6 = r4 × r5**
   - **[r2] = r6** (stores initial cost in)
4. **Check discount condition**
   - CMP r5, #10
   - BLE NO_DISCOUNT (if **days ≤ 10**, skip discount)
5. **Compute discount when days > 10**
   - r7 = 5
   - r7 = r7 × r6 → r7 = cost × 5
   - r4 = 100
   - r7 = r7 / r4 using SDIV → r7 = (cost×5)/100
6. **Apply discount**
   - r6 = r6 - r7
   - [r2] = r6 (store final discounted cost)
7. **Return**
   - Restore registers and return to caller

7. Module Output (in memory)

After **Calc_Room_Rent** runs:

- Always writes base cost first:
  - **ROOM_COST = ROOM_RATE × DAYS_STAYED**
  - If **DAYS_STAYED > 10**, overwrites with discounted final cost:
  - **ROOM_COST = (ROOM_RATE × DAYS_STAYED) – ((ROOM_RATE × DAYS_STAYED × 5) / 100)**

9. Output (Examples)

Example 1: No discount (≤ 10 days)

Assume:

- ROOM_RATE = 200
- DAYS_STAYED = 7

Base cost: ROOM_COST = 200 × 7 = 1400

Days ≤ 10 ⇒ no discount

Final output:**ROOM_COST = 1400**

Example 2: Discount applies (> 10 days)

Assume:

- ROOM_RATE = 200
- DAYS_STAYED = 12

Base cost: cost = 200 × 12 = 2400

Discount:discount = (2400 × 5) / 100 = 120

Final:ROOM_COST = 2400 − 120 = 2280

Final output:**ROOM_COST = 2280**

Example 3: Integer truncation case

Assume:

- ROOM_RATE = 333
- DAYS_STAYED = 11

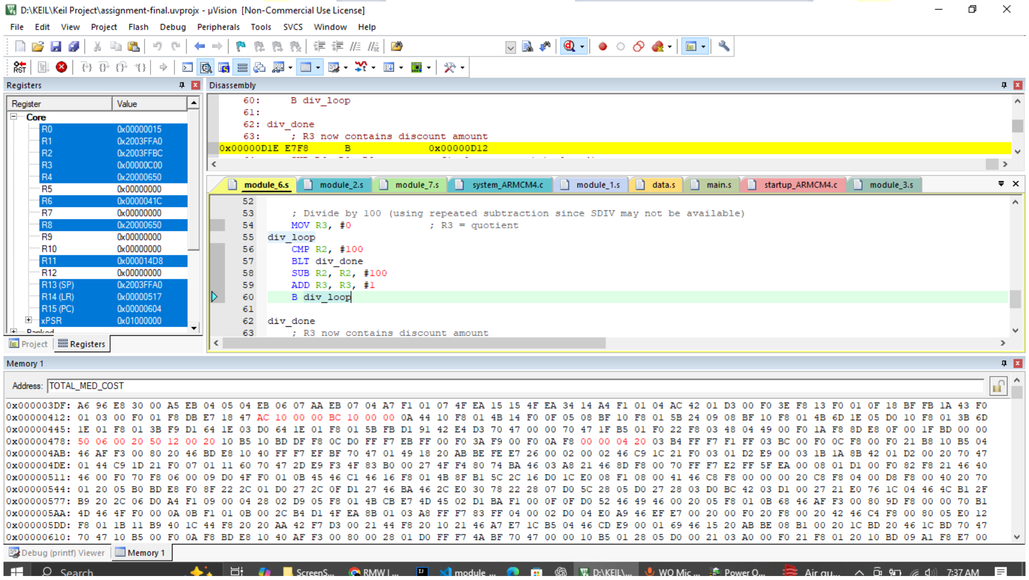
Base: cost = 333 × 11 = 3663

Discount:(3663×5)/100 = 18315/100 = 183 **(fraction discarded)**

Final:ROOM_COST = 3663 − 183 = 3480

Final output: **ROOM_COST = 3480**

10. Output Screenshot:



MODULE 7: Medicine Billing Module (module_7.s)

1. Objective

Compute the **total medicine bill** for a patient from a list of medicines.
Each medicine entry contains: **unit_price,quantity,days**

For each entry compute: med_cost = unit_price × quantity × days

2. Data Used

Input

- **MED_LIST** : list of medicines, each medicine uses **3 words (12 bytes)**
    - word 0: unit_price
    - word 1: quantity
    - word 2: days
- **NUM_MEDS** : number of medicines in the list

Output

- **TOTAL_MED_COST** : final total medicine cost stored in memory

2. Test Data Used (from MED_LIST)

Medicine entries (unit_price, quantity, days):

1. (10, 2, 3) → cost = 10×2×3 = 60
2. (50, 1, 5) → cost = 50×1×5 = 250
3. (20, 3, 2) → cost = 20×3×2 = 120

Expected total: **60+250+120=430**

Final expected output: **TOTAL_MED_COST = 430**

3. Solution Technique

Main Driver

1. Load base pointer of medicine list:
    - R0 = &MED_LIST
2. Load number of medicines:
    - R1 = [NUM_MEDS]
3. Load output address:
    - R2 = &TOTAL_MED_COST
4. Call Compute_Medicine_Bill
5. Stop in an infinite loop for debugging

4. Billing Computation (Compute_Medicine_Bill)

1. Initialize:
    - total_cost = 0
    - loop index i = 0
2. Loop while i < NUM_MEDS:
    - Compute address of entry i:
        - each entry size = **12 bytes**
        - entry_addr = MED_LIST + (i × 12)
    - Load the 3 fields: price, qty, days
    - Compute med_cost = price × qty × days
    - Add to running total
    - Increment i
3. Store final total_cost into TOTAL_MED_COST

This uses **array traversal with indexed addressing** and **multiplication-based cost calculation**.

5. Inputs Passed (Register Passing)

Before calling Compute_Medicine_Bill:

- R0 = &MED_LIST
- R1 = NUM_MEDS
- R2 = &TOTAL_MED_COST

6. Step-by-step execution logic

1. **Initialize totals**
    - r7 = 0 (total_cost)
    - r4 = 0 (i)
2. **Loop condition**
    - If i >= NUM_MEDS → exit loop
3. **Compute address of i-th medicine entry**
    - offset = i × 12
    - entry_ptr = MED_LIST + offset
4. **Load fields**
    - unit_price = [entry_ptr + 0]
    - quantity = [entry_ptr + 4]
    - days = [entry_ptr + 8]
5. **Compute cost**
    - med_cost = unit_price × quantity × days
6. **Accumulate**
    - total_cost += med_cost
7. **Increment and repeat**
    - i = i + 1
8. **Store output**
   [TOTAL_MED_COST] = total_cost

7. Module Output (What changes in memory)

After **Compute_Medicine_Bill** finishes:

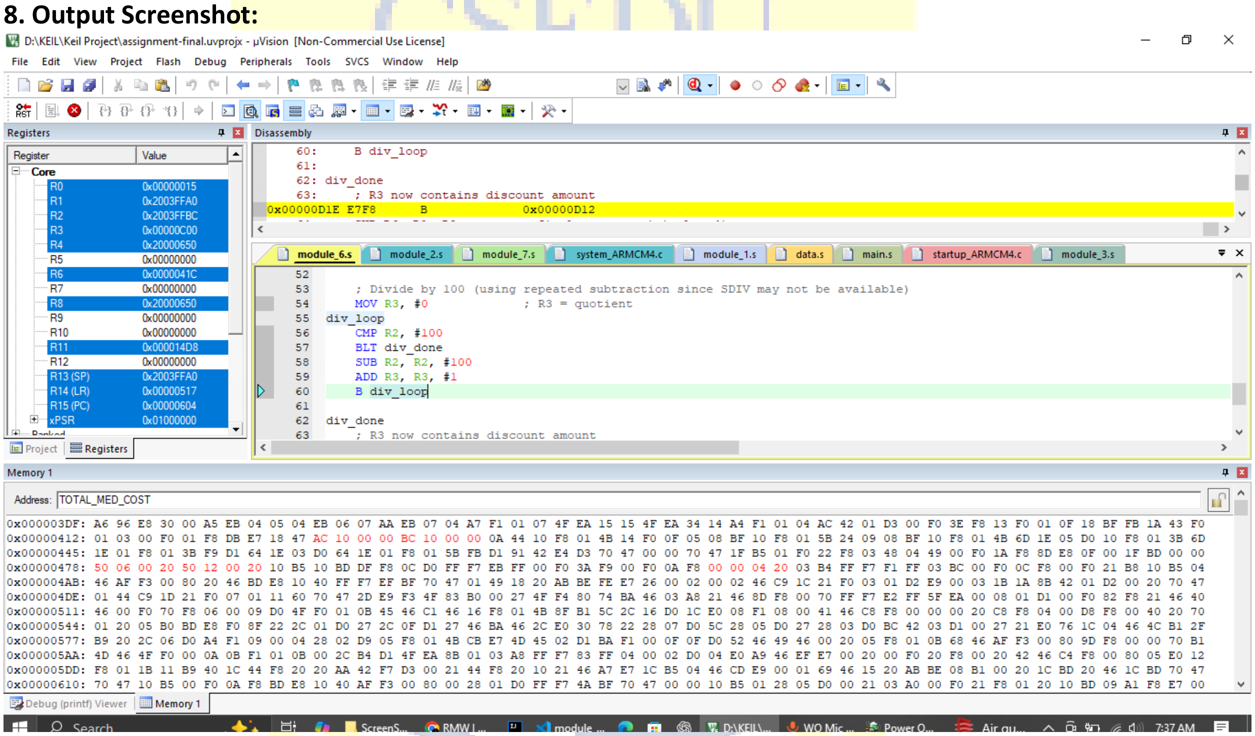
- **TOTAL_MED_COST** contains the sum of all medicine costs.

For the given list:

- med1 = 60
- med2 = 250
- med3 = 120

Output:**TOTAL_MED_COST = 430**

8. Output Screenshot:



MODULE 8: Patient Bill Aggregator (module_8.s)-

1. Objective

Compute the patient's final payable bill by summing:

total_bill = treatment + room + medicine + lab_tests

Also perform overflow checking during addition. If overflow occurs:

- Set ERROR_FLAG = 1
- Store TOTAL_BILL = 0 (safe output)

2. Data Used (RAM Variables)

Inputs (loaded from memory)

- **TREATMENT_COST**
- **ROOM_COST**
- **MEDICINE_COST**
- **LABTEST_COST**

Outputs

- **TOTAL_BILL :** final aggregated bill (or 0 on overflow)
- **ERROR_FLAG :** overflow indicator
    - 0 = no overflow
    - 1 = overflow occurred

3. Solution Technique (logic pattern)

Main Driver

1. Directly calls **Compute_Total_Bill**
2. Stops execution in an infinite loop for debugging

4. Bill Aggregation + Overflow Detection (Compute_Total_Bill)

1. Clear error flag (**ERROR_FLAG = 0**)
2. Load all cost components from memory into registers
3. Perform sequential additions using ADDS:
    - ADDS updates condition flags
    - If carry flag (C) is set after addition → overflow for unsigned sum (too large for 32-bit)
4. If overflow detected at any step:
    - **Set ERROR_FLAG = 1**
    - **Set TOTAL_BILL = 0**
5. If no overflow:
    - Store final sum into **TOTAL_BILL**

5. Step-by-step execution logic

1. **Initialize / clear overflow flag**
   - **r7 = 0**
   - **[ERROR_FLAG] = 0**
2. **Load all input costs**
   - **r1 = [TREATMENT_COST]**
   - **r2 = [ROOM_COST]**
   - **r3 = [MEDICINE_COST]**
   - **r4 = [LABTEST_COST]**
3. **Start total at 0**
   - **r5 = 0**
4. **Add treatment + room**
   - **r5 = r1 + r2 using ADDS**
   - **If BCS overflow → overflow handler**
5. **Add medicine**
   - **r5 = r5 + r3 using ADDS**
   - **If BCS overflow → overflow handler**
6. **Add lab tests**
   - **r5 = r5 + r4 using ADDS**
   - **If BCS overflow → overflow handler**
7. **No overflow case (normal store)**
   - **[TOTAL_BILL] = r5**
   - **Return**

6. Overflow handling logic

If overflow occurs at any add:

1. **Set overflow indicator:**
   - **[ERROR_FLAG] = 1**
2. **Force safe output:**
   - **[TOTAL_BILL] = 0**
3. **Return**

7. Module Output

If no overflow:

- ERROR_FLAG = 0
- TOTAL_BILL = TREATMENT_COST + ROOM_COST + MEDICINE_COST + LABTEST_COST

If overflow occurs:

- ERROR_FLAG = 1
- TOTAL_BILL = 0

7. Output

Example 1: Normal case (no overflow)

Assume:

- TREATMENT_COST = 600
- ROOM_COST = 2280
- MEDICINE_COST = 430
- LABTEST_COST = 200

Total:600+2280+430+200=3510

Output:

- **TOTAL_BILL** = 3510
- **ERROR_FLAG** = 0

Example 2: Overflow case (sum too large for 32-bit unsigned)

Assume (example values near the 32-bit limit):

- **TREATMENT_COST** = 0xFFFFFFF0
- **ROOM_COST** = 0x00000040
- **others = 0**

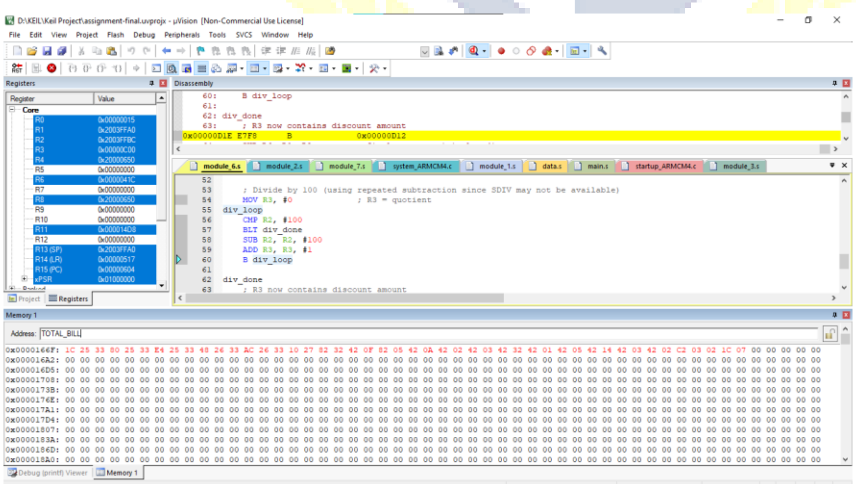
First add:

- **0xFFFFFFF0 + 0x40 = 0x100000030 → exceeds 32-bit → carry set**

Output:

- **TOTAL_BILL = 0**
- **ERROR_FLAG = 1**

8.Output Screenshot:



MODULE 9: Sorting Patients by Criticality (module_9.s)-

1. Objective

Prioritize ICU triage by sorting patients in **descending order of alert count** (highest alerts = most critical).
This module sorts both:
• patient_alert_countX (the key for sorting), and
• the **patient identifier value stored at PATIENTx_ADDR**, so the identifier stays matched with its alert count.

2. Data Used (RAM Variables)

Inputs (from memory)

• **patient_alert_count1, patient_alert_count2, patient_alert_count3**
• **PATIENT1_ADDR, PATIENT2_ADDR, PATIENT3_ADDR** (each holds the patient identifier value)

Outputs

• Sorted **patient_alert_count1, patient_alert_count2, patient_alert_count3**
• Sorted identifier values written back to **PATIENT1_ADDR, PATIENT2_ADDR, PATIENT3_ADDR**

3. After sorting:

- slot 1 contains most critical
- slot 3 contains least critical

4. Solution Technique

Sorting Method: Bubble Sort (Descending Order)

1. Load the three alert counts into registers **(R6–R8)**
2. Load the corresponding patient identifier values into registers **(R9–R11)**
3. Repeatedly compare adjacent pairs and swap if out of order:
   o Compare **(count1, id1)** with **(count2, id2)**
   o Compare **(count2, id2)** with **(count3, id3)**
4. Perform multiple passes using an outer loop until sorted
5. Store the sorted counts and identifiers back to memory

This ensures the **alert count and its corresponding patient identifier are always swapped together**.

5. Register Usage Summary

• **Alert Counts:**
o R6 = count1, R7 = count2, R8 = count3

• **Patient Identifiers:**
o R9 = id1, R10 = id2, R11 = id3

• **Temporary Register for Swapping:**
o R2

• **Outer Loop Counter:**
o R12

6. Step-by-step sorting logic

1) Load from Memory

• Load addresses of alert counts and patient identifier locations
• Load values:
o R6 = [patient_alert_count1]
o R7 = [patient_alert_count2]
o R8 = [patient_alert_count3]
o R9 = [PATIENT1_ADDR]
o R10 = [PATIENT2_ADDR]
o R11 = [PATIENT3_ADDR]

2) Bubble Sort Passes

Outer loop runs **n – 1 = 2 passes** (for n = 3).

Pass Logic (Inner Comparisons)

A. Compare First Pair
• CMP R6, R7
• If R6 < R7 (wrong order for descending):
o Swap counts: R6 ↔ R7
o Swap identifiers: R9 ↔ R10

B. Compare Second Pair
• CMP R7, R8
• If R7 < R8:
o Swap counts: R7 ↔ R8
o Swap identifiers: R10 ↔ R11

This moves the highest alert count toward the first position while keeping identifiers aligned.

3) Store Sorted Results Back to RAM

After sorting completes:
• [patient_alert_count1] = R6
• [patient_alert_count2] = R7
• [patient_alert_count3] = R8
• [PATIENT1_ADDR] = R9
• [PATIENT2_ADDR] = R10
• [PATIENT3_ADDR] = R11

7. Module Output

After module_nine returns:
• Alert counts are arranged in **descending order**
• Patient identifier values are rearranged to match the new order.So memory slots represent triage ranking:

1. (count1, id1) = **most critical**
2. (count2, id2) = **medium critical**
3. (count3, id3) = **least critical**

8. Output Example

Given Alert Counts

• Patient 1: count = 2
• Patient 2: count = 1
• Patient 3: count = 3

Sorting Result (Descending)

1. (3, Patient 3) — most critical
2. (2, Patient 1)
3. (1, Patient 2) — least critical

Final Memory State

• patient_alert_count1 = 3, [PATIENT1_ADDR] = Patient3
• patient_alert_count2 = 2, [PATIENT2_ADDR] = Patient1
• patient_alert_count3 = 1, [PATIENT3_ADDR] = Patient2

MODULE 10: UART Summary Report Generator (module_10.s)-

1. Objective

Send a formatted **ICU patient summary** to a serial/debug terminal using **byte-by-byte character output** (through debug_send).
This report includes (Patient 1 only):

- Patient ID
- Patient Name
- Age
- Ward
- Latest vitals (HR, BP, O2)
- Total alerts
- Total bill (formatted as dollars.cents)

2. Data Used

Inputs (imported from other modules)

- Patient 1 data:
  - patient1_name
  - patient_id1
  - patient1_age
  - patient1_ward
  - patient_alert_count1
  - HR1_data, BP1_data, O21_data

- Billing data:
  - TOTAL_BILL

- Output routine:
  - debug_send (sends one character to terminal)

Output (to terminal)

- A human-readable summary transmitted **character-by-character** using debug_send.

3. Solution Technique (logic pattern)

Step 1: Print ICU Header

- print_icu_header prints:

1. "ICU PATIENT SUMMARY"
2. A newline
3. A separator line of '-' repeated **22 times**
4. A newline

Step 2: Print Patient 1 Summary

• print_patient1_summary prints in order:

1. ID
2. Name
3. Age
4. Ward
5. HR
6. BP
7. O2
8. Alerts
9. Bill (TOTAL_BILL in dollars.cents)
10. Separator line (22 dashes)

Step 3: Convert integers to ASCII

This module includes conversion/printing routines:

• print_string
o Reads a null-terminated string using LDRB
o Sends each character using debug_send

• print_number
o Converts and prints an integer in **decimal ASCII**
o Supports up to 3 digits clearly (0–999) using manual division

• simple_divide
o Performs division by repeated subtraction
o Outputs: quotient in R0, remainder in R1

• print_bill_amount
o Prints billing amount in **dollars.cents**
o Formula:

- dollars = TOTAL_BILL / 100
- cents = TOTAL_BILL % 100
  o Prints decimal point .
  o Prints cents with **two digits** (leading zero if needed)

Step 4: Transmit byte-by-byte

• All output ultimately goes through:
o debug_send
Each character is transmitted **one at a time**, satisfying UART-style byte transmission.

4. Step-by-step Execution Flow

1. BL print_icu_header

2. BL print_patient1_summary
3. Return to caller (BX LR)

Header Section

1. "ICU PATIENT SUMMARY"
2. "\r\n"
3. "----------------------" (22 dashes)
4. "\r\n"

Patient 1 Summary Lines (in order)

• "ID: " + patient_id1 (decimal) + "\r\n"
• "Name: " + patient1_name + "\r\n"
• "Age: " + patient1_age (decimal) + "\r\n"
• "Ward: " + patient1_ward (decimal) + "\r\n"
• "HR: " + HR1_data (decimal) + "\r\n"
• "BP: " + BP1_data (decimal) + "\r\n"
• "O2: " + O21_data (decimal) + "\r\n"
• "Alerts: " + patient_alert_count1 (decimal) + "\r\n"
• "Bill: $" + TOTAL_BILL (dollars.cents) + "\r\n"
• Separator line again (22 dashes) + newline

Note: Patient ID is printed in **decimal**, not hex, because the code calls print_number

6. Module Output

This module does not modify patient or billing memory values. Its output is purely:
• Formatted summary characters transmitted to terminal via debug_send.

7. Example Output (Format Example)

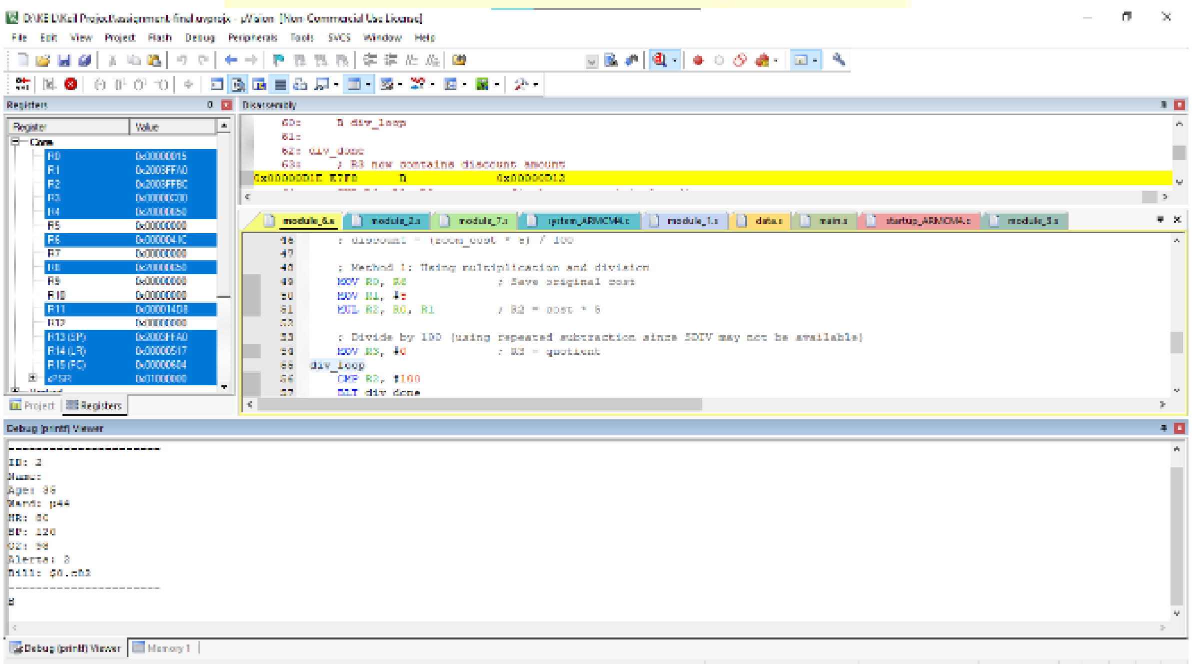
If:
• patient_id1 = 25
• patient1_name = "Ali"
• patient1_age = 45
• patient1_ward = 3
• HR1_data = 130
• BP1_data = 170
• O21_data = 85
• patient_alert_count1 = 2
• TOTAL_BILL = 313000 (=$3130.00)

Then terminal shows:

ICU PATIENT SUMMARY

ID: 25
Name: Ali
Age: 45
Ward: 3
HR: 130
BP: 170
O2: 85
Alerts: 2
Bill: $3130.00

7.Output Screenshot:



MODULE 11: Anomaly Detection & Safety Check (module_11.s)-

1. Objective

Detect unsafe or abnormal system conditions during runtime and raise a system-wide error.
This module performs **multiple safety checks** across sensors, medicine scheduling, memory usage, and billing sanity.

The module checks for:

1. **Sensor malfunction** (same sensor value repeated more than 10 times)
2. **Invalid medicine dosage** (zero or negative interval values)
3. **Memory boundary violations** (addresses outside valid RAM range)
4. **Billing sanity errors** (negative, overflow, or unreasonably large bill values)

If any anomaly is detected:
• ERROR_FLAG is set to 1
• An **error record** is generated and reported via debug output (timestamped)

2. Data Used (RAM Variables)

Inputs (from memory)

Sensor Buffers (circular buffers, size = 10)

• hr1_buffer, hr2_buffer, hr3_buffer
• bp1_buffer, bp2_buffer, bp3_buffer
• o21_buffer, o22_buffer, o23_buffer

Sensor Indices

• hr1_index, hr2_index, hr3_index
• bp1_index, bp2_index, bp3_index
• o21_index, o22_index, o23_index

Medicine Scheduling

• MED1_INTERVAL, MED2_INTERVAL, MED3_INTERVAL

Billing

• TOTAL_BILL

System & Debug

• timestamp_counter
• debug_send

Outputs

• ERROR_FLAG : global error indicator
o 0 = no error
o 1 = error detected

• Debug output stream containing formatted error records
(sent via debug_send)

3. Solution Technique

Main Routine (module_eleven)

1. Clear any previous error:
   • ERROR_FLAG = 0
2. Execute all safety checks in sequence:
   1. Sensor malfunction detection
   2. Medicine dosage validation
   3. Memory boundary checking
   4. Billing sanity checking
3. Send a completion marker 'B' via debug output
4. Return to caller

4. Safety Checks Performed

Check 1: Sensor Malfunction Detection

Goal: Detect frozen sensors producing the same value repeatedly.

Logic:

• All **9 sensor buffers** are checked
(3 patients × HR, BP, O2)

For each sensor:

1. Ensure at least 10 readings exist
2. Compare the **last 10 readings**
3. If all 10 readings are identical → sensor malfunction

Implementation:

• check_sensor_repetition
• Calls check_single_sensor for each buffer
• Uses circular buffer logic with index wrapping

Condition for error:

• Same value repeated **10 times**

Error action:

• Error code = 1 (sensor malfunction)
• Record includes:
o Sensor type (HR / BP / O2)
o Patient number
o Timestamp

Check 2: Invalid Medicine Dosage

Goal: Ensure medicine intervals are valid.

Logic:

1. Load MEDx_INTERVAL
2. If interval ≤ 0 → invalid dosage

Checked:

• MED1_INTERVAL
• MED2_INTERVAL
• MED3_INTERVAL

Error action:

• Error code = 2 (medicine error)
• Item field = medicine number (1, 2, or 3)

Check 3: Memory Boundary Violation

Goal: Ensure critical data resides within valid RAM.

Valid RAM range:

• Start: 0x20000000
• End: 0x20010000

Checked addresses include:

• Sensor buffers
• patient_alert_count1
• TOTAL_BILL
• MED1_INTERVAL

Logic:

• If address < start OR address > end → error

Error action:

• Error code = 3 (memory boundary error)

Check 4: Bill Sanity Check

Goal: Detect billing corruption or overflow.

Conditions checked:

1. Bill must not be negative
2. Bill must not exceed **$1,000,000** (100,000,000 cents)
3. Bill must not exceed 0x7FFFFFFF

Error codes:

• 4 → negative bill
• 5 → bill too large
• 6 → bill overflow

5. Error Recording Mechanism (record_error)

When an error is detected:

1. ERROR_FLAG is set to 1
2. Current timestamp is loaded from `timestamp_counter`
3. Error is reported via debug output

Debug format: “E=<error_code>:<item>:<timestamp>”

6. Step-by-Step Execution Flow

1. ERROR_FLAG ← 0
2. check_sensor_repetition
3. check_medicine_zero
4. check_memory_boundaries

5. check_bill_sanity
6. Send 'B' via debug_send
7. Return (BX LR)

7. Module Output

If NO anomaly is detected:

• ERROR_FLAG = 0
• No error messages sent

If ANY anomaly is detected:

• ERROR_FLAG = 1
• One or more error records sent via debug output

8. Example Execution Scenario

Given:

• HR buffer contains 10 identical values
• MED2_INTERVAL = 0
• TOTAL_BILL = 120000000 (too large)

Execution:

1. Sensor malfunction detected → error code 1
2. Medicine interval invalid → error code 2
3. Bill too large → error code 5

Final state:

• ERROR_FLAG = 1
• Multiple error records printed with timestamps

8.Output Screenshot: