2GA3 Tutorial #5

DATE: October 22nd, 2021

TA: Jatin Chowdhary

T01

- I gotta kick you out by 10:20AM
  - But only this time
- Someone's feedback was: ‘‘more energy’’
  - Can do!
    - It's time to double down

T02

- We didn’t have enough time for feedback last class
  - But we’re doing it right now!
- Grab a sheet, something to write with, and fill it out
  - Once you finish, toss it in the binder
    - And then get whatever snack you want

Changes

- Going forward, tutorials will be structured in the following manner:
  - 5 MINS
    - General stuff (i.e. Administrative)
  - 5 MINS
    - Review (i.e. Multiple choice questions)
  - 20 MINS
    - Easy questions
  - 20 MINS
    - Hard questions

Administrative Stuff (1)

- How was the midterm?
  - Easy, medium, hard, etc.
  - MSAF?
  - Are there any questions I should review?
- ‘‘Is there a solution manual?’’
  - I’ll try to find one
- Will take assignment questions now
  - Fair warning: I’ll be useless
- Get out of your comfort zone
  - I’ve got 5 weeks to work on you
    - Why? (I’ll explain later)

Administrative Stuff (2)

- Be mindful of what you eat or put in your mouth

Review Question #1

- **Question:** Can we use a base-1 (unary) number system to represent all the numbers? Assume that *special* tricks are not used.
- **Options:**
  - A) Yes we can
  - B) No we cannot
  - C) Depends on the number
  - D) Only in *CalcCheck*
  - E) None of the above

Explanation For #1

- **Explanation:** In a base-1 number system, we only have 1 number to work with. For example:
  - $1^0 + 1^1 + 1^2 + 1^3 + \ldots + 1^n$
    - We can represent all the natural numbers, except for 0
      - A ‘‘trick’’ is needed to represent negative numbers
        - Expect compromises
  - $2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^n$
    - Same as above
- *Note: We can have a voltage difference in a unary number system, as long as the base-number isn’t 0. And with a few tricks, we can use a unary number system in a computer*
  - *i.e.* The following wouldn’t work: $0^0 + 0^1 + 0^2 + 0^3 + \ldots + 0^n$
    - *This stuff is beyond the content of 2GA3; so don’t worry about it*
      - *But it is good to know*

Review Question #2

- **Question:** Find the typo: *addi x10, x10, x4*
- **Options:**
  - A) *addi*
  - B) *x10*
  - C) *x4*
  - D) The commas
  - E) The instruction needs to be capitalized
  - F) None of the above

Review Question #3

- **Question:** What does *blt* stand for
- **Options:**
  - A) Bacon, Lettuce, Tomatoes
  - B) Branch Less Than
  - C) BaselLand Transport
  - D) Bounded Linear Transformation
  - E) None of the above
  - F) All of the above

Review Question #4

- **Question:** Fill in the blanks: RISC-V stands for _____, and is _____ endian
- **Options:**
  - A) *Reduced instructions on system and computer* **&&** *Little*
  - B) *Reduced instruction set compiler* **&&** *Big*
  - C) *Reduced instruction set compiler* **&&** *Little*
  - D) *Reduced instruction set computer* **&&** *Big*
  - E) *Reduced instruction set computer* **&&** *Little*
  - F) None of the above

Review Question #5

- **Question:** Which of the following correspond to *floating point* instructions?
- **Options:**
  - A) flw
  - B) fld
  - C) fsw
  - D) fsd
  - E) fadd.s
  - F) fmul.d
  - G) feq.s
  - H) All of the above
  - I) None of the above

Review Question #5

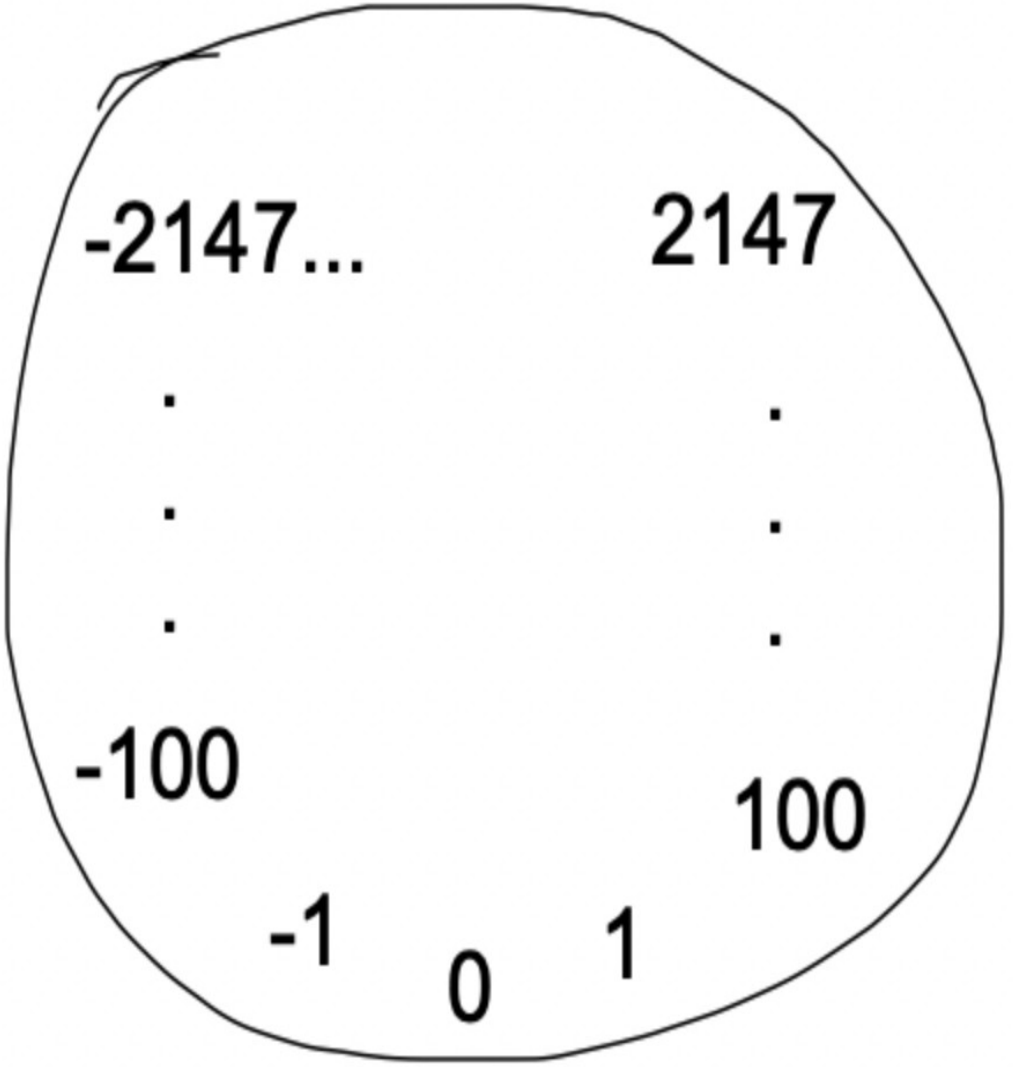
- **Question:** Why do we have immediate instructions?
  - Asked By Esha
- **Answer:** Because it's quicker
  - Loading from memory is expensive (time consuming)
    - i.e. It's easier to bake a cake if all of the ingredients are in the pantry. But if you have to go to the store, then it's going to slow you down a lot
    - i.e. *Pen demo*

Clockwork Arithmetic

- Clockwork arithmetic
  - Also known as **mod**ular arithmetic
    - This is where the **mod** function comes from (i.e. 9 **mod** 2 = 1)
- Refer to the picture:
  - 1:00PM + 2 HRs = ? PM
    - No *flow
  - 7:00PM – 5 HRs = ? PM
    - No *flow
  - 9:00PM + 6 HRs = ? ? ?
    - Overflow or underflow?
  - 6:00AM – 12 HRs = ???
    - Overflow or underflow?

Overflow In C

- *"Talk is cheap, show me the code" – Linus T.*
  - *i.e. *flow.c*
- Summary:
  - If the value is too big, then overflow
  - If the value is too small, then underflow



Tutorial Question #1

- **Question:** Assume $(185)_{10}$ and $(122)_{10}$ are unsigned 8-bit decimal integers. *Calculate 185 – 122.* Is there underflow, overflow, or neither?
- **Precursor:**
  - 185 and 122 are decimal integers
    - They are 8-bits
      - This is the size
    - Unsigned
      - What does this mean?

Answer For Question #1

- **1.** 8 bit signed means that:
  - The maximum value is:
    - $2^8 - 1 = 255$
  - The minimum value is:
    - *0*
- **2.** Convert $(185)_{10}$ and $(122)_{10}$ to binary
  - See #3
- **3.** The conversion is:
  - $(185)_{10} = (1011\ 1001)_2$
  - $(122)_{10} = (0111\ 1010)_2$
- **4.** Subtract the result
  - $1011\ 1001 - 0111\ 1010$

    $= (0011\ 1111)_2$

    $= (63)_{10}$
- **5.** 63 is within [0, 255]
  - Hence, there is no overflow

Tutorial Question #2

- **Question:** Assume $(185)_{10}$ and $(122)_{10}$ are signed 8-bit decimal integers stored in *sign-magnitude format*. *Calculate (185 + 122)*. Is there overflow, underflow, or neither?
- **Precursor:**
  - What is signed?
    - How does this affect the 8-bit size?
      - What is the range? (Min/Max)
  - What is sign-magnitude format?
    - The most significant digit (MSD) determines if the number ispositive/negative
      - MSD = Leftmost digit

Answer For Question #2

- $(122)_{10} = (\mathbf{0}111\ 1010)_2$
  - The first bit is **0**, so this is a positive number
    - So the value doesn't change
- $(185)_{10} = (\mathbf{1}011\ 1001)_2$
  - The first bit is **1**, so this is a negative number
    - So the number is $(0011\ 1001)_2 = 57$
      - But, because of the sign bit, it is negative
        - $(\mathbf{1}011\ 1001)_2 = -57$
          - *MSD IS FOR THE SIGN (IN A SIGNED NUMBER)*
- Calculation
  - Next slide

Answer For Question #2

- The equation is: –57 + 122
  - Answer: 65
    - In binary: $(0111\ 1010)_2 - (0011\ 1001)_2 = (0100\ 0001)_2$

      122 57 65
- But wait!
  - What's the range (min/max) of a signed 8-bit number in signed-magnitude format?
    - Answer: [-127, 127]
      - Does 65 fall into this range? **Yes**
        - Hence, there is overflow/underflow/**neither**

Review Question #7

- **Question:** In an *unsigned* number, the *MSD* represents the signage; whether it is positive or negative?
- **Options:**
  - True
  - False
  - I'm Hungry

Review Question #8

- **Question:** In *signed-magnitude format*, how many ways can *0* be represented?
- **Options:**
  - 1
  - 2
  - 3
  - 4
  - 0
  - ∞
  - None of the above

2 Ways?!?

- We know that $(0)_{10}$ is $(0000\ 0000)_2$
- But, recall that the most significant bit is ONLY for signage.
  - Since the MSD doesn't affect the numeric value, other than signage, we can add a 1:
    - $(1000\ 0000)_2$
      - And this is still 0

Tutorial Question #3

- **Question:** Assume 185 and 122 are *signed* 8-bit decimal integers stored in *sign-magnitude format*. Calculate *185 −122*. Is there overflow, underflow, or neither?
- **Answer:** (Same logic as question #2)
  - The numbers are:
    - $(185)_{10} = (1011\ 1001)_2 = (-57)_{10}$
    - $(122)_{10} = (0111\ 1010)_2 = (122)_{10}$
  - Calculate: (185 – 122) → (–57) – (122)
    - –57 – 122 = –179
  - Overflow/underflow/neither?
    - Next slide

Answer For Question #3

- What's the range for a signed 8-bit decimal in *signed-magnitude format*?
  - (It's the same as the previous question)
    - It is: [–127, 127]
- The answer we got is: –179
  - Does –179 fall into [–127, 127]
- Is there overflow, underflow, neither?
  - Answer: Overflow

Saturation Arithmetic (1)

- All operations are limited to a fixed range
  - In other words, you cannot go above the maximum value, or below the minimum value
    - If you reach the maximum value, you stay at the maximum value
    - If you reach the minimum value, you stay at the minimum value
- For example, assume we are working with an 8-bit *unsigned* integer
  - Max value = 255
  - Min value = 0
- You don’t wrap around like Clockwork arithmetic

Saturation Arithmetic (2)

- For example, assume we are working with an 8-bit *signed* integer in *signed-magnitude format*
  - Max value = 127
  - Min value = –127
- You don’t wrap around like Clockwork arithmetic
  - Instead, you stay at the min/max

Two's Complement (1)

- What is it?
  - A way of storing integers so that common math problems are simple to implement
- Rules:
  - **Zero**
    - Represented by all 0's
  - **Positive Integers**
    - Start counting as you normally would
  - **Negative Integers**
    - Do the same thing as above, but then flip the bits
      - 0 → 1
      - 1 → 0

Two's Complement (2)

- Positive numbers:
  - $(0000)_2 = (0)_{10}$
  - $(0001)_2 = (1)_{10}$
  - $(0010)_2 = (2)_{10}$
  - $(0011)_2 = (3)_{10}$
- Negative numbers:
  - $(1111)_2 = (-1)_{10}$
  - $(1110)_2 = (-2)_{10}$
  - $(1101)_2 = (-3)_{10}$
  - $(1100)_2 = (-4)_{10}$

Two's Complement (3)

- Negative numbers:
  - $(1111)_2 = (-1)_{10}$
    - Another way to think about this:
      - $-8 + 4 + 2 + 1 = -1$
  - $(1110)_2 = (-2)_{10}$
    - $-8 + 4 + 2 = -2$
  - $(1101)_2 = (-3)_{10}$
  - $(1100)_2 = (-4)_{10}$

Tutorial Question #4

- **Question:** Assume 151 and 214 are *signed* 8-bit integers stored in *two's complement*. Calculate 151 + 214 using *saturating arithmetic*.

  The result should be written in decimal.

- **Answer:**
  - $(151)_{10} = (1001\ 0111)_2$
    - $(1001\ 0111)_2 \rightarrow -128 + 16 + 4 + 2 + 1 = -105$
  - $(214)_{10} = (1101\ 0110)_2$
    - $(1101\ 0110)_2 \rightarrow -128 + 64 + 16 + 4 + 2 = -42$
  - $-105 + (-42) = -147$
    - But wait! We are using saturation arithmetic
      - So what do we do? *Next slide*

Answer For Question #4

- **Answer:**
  - What is the range of values in a signed 8-bit number stored in two's complement?
    - Answer: [–128, 127]
  - The result from the previous slide is –147
    - Since we cannot go past –128, the answer is **–128**
      - *This is saturation arithmetic; we cannot go higher than the max, or lower than the min*

Hold On… -127 VS. -128

- Recall that:
  - In *signed-magnitude format*, the range of possible values is [–127, 127]
  - In *two's complement*, the range of possible values is [–128, 127]
- Why?
  - Because in *signed-magnitude format*, there are 2 ways of representing 0
    - This problem doesn't exist in *two's complement*

Tutorial Question #5

- **Question:** Assume 151 and 214 are *signed* 8-bit decimal integers stored in *two's complement format*. *Calculate (151 − 214)* using saturating arithmetic.

  The result should be written in decimal.

- **Answer:** *(Same logic as question #4)*
  - Convert to binary:
    - $(151)_{10} = (1001\ 0111)_2$
    - $(214)_{10} = (1101\ 0110)_2$
  - Apply two's complement:
    - *Next slide*

Answer For Question #5

- **Answer:**
  - Apply two's complement:
    - $(1001\ 0111)_2$ → $-128 + 16 + 4 + 2 + 1 = \mathbf{(-105)_{10}}$
    - $(1101\ 0110)_2$ → $-128 + 64 + 16 + 4 + 2 = \mathbf{(-42)_{10}}$
  - Calculate:
    - $(-105) - (-42) = -105 + 42 = (-63)_{10}$

Tutorial Question #6

- **Question:** Assume 151 and 214 are *unsigned* 8-bit integers. *Calculate 151 + 214* using *saturating arithmetic*. The result should be written in decimal.
- **Homework!**
  - Very simple
    - *Hint:* There is saturation/overflow!
    - *Hint:* Pay attention to the format of the number
      - *See tutorial question #1*

What Is All This?

- What is the point of clockwork/modular arithmetic? Saturation arithmetic?
  - A way for the computer to do math
- What is 1's complement? 2's complement? Sign-magnitude format?
  - A way for the computer to represent numbers

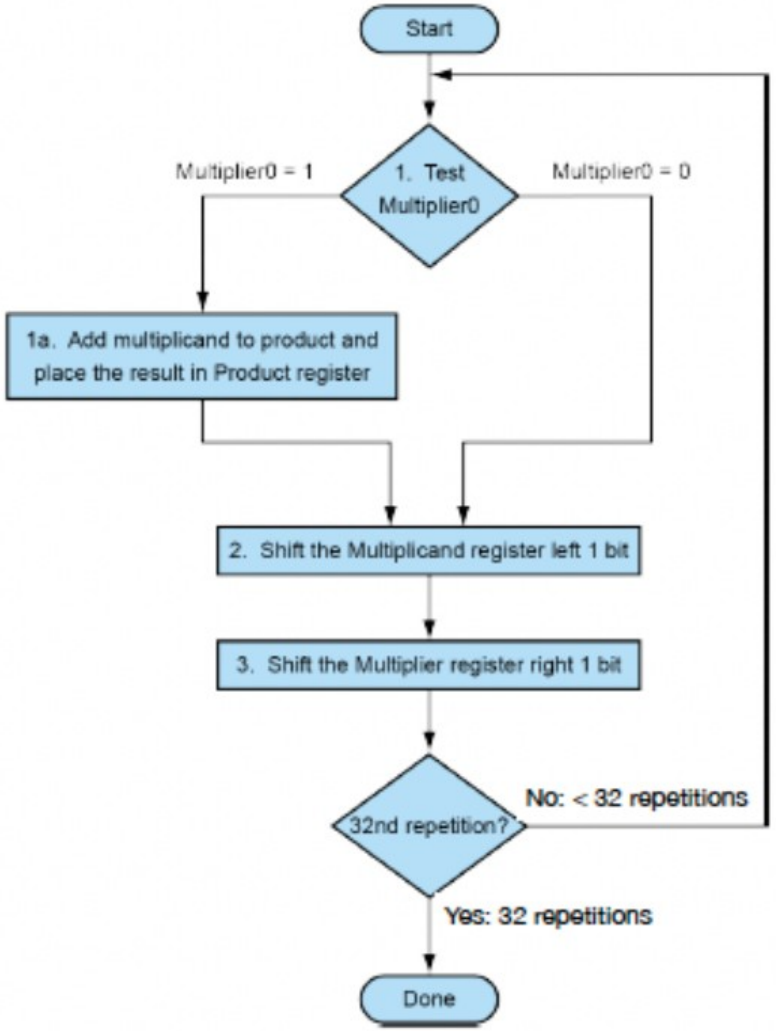
Quick Note

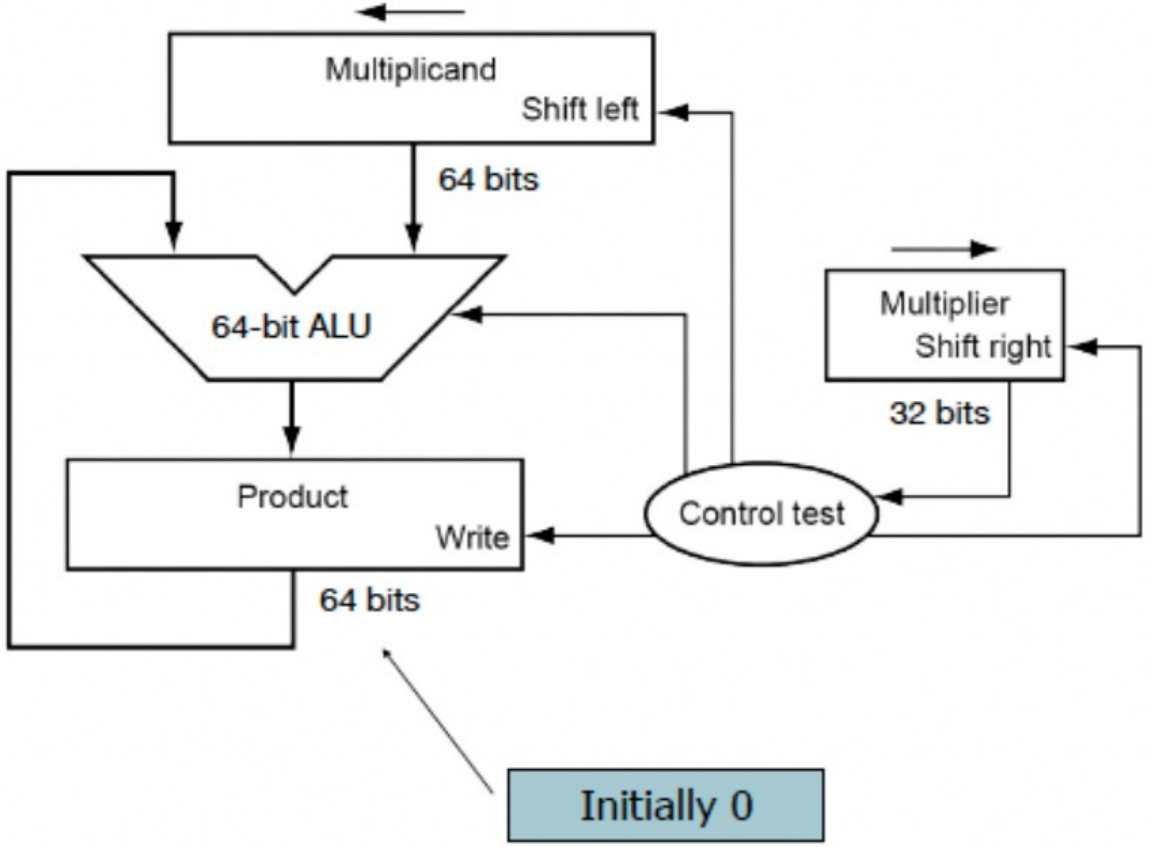
- Always pay attention to the format of the number
  - Is it in two's complement? Signed-magnitude? One's complement?
    - This makes a big difference!
      - Also, is it signed/unsigned?

Tutorial Question #7

- **Question:** Calculate the time necessary to perform a multiply using the approach given in Figure below if an integer is 8 bits wide and each step of the operation takes four time units. Assume that in step 1a an addition is always performed—either the multiplicand will be added, or a zero will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case.
  - *Figure on next slide*

Figure For Question #7





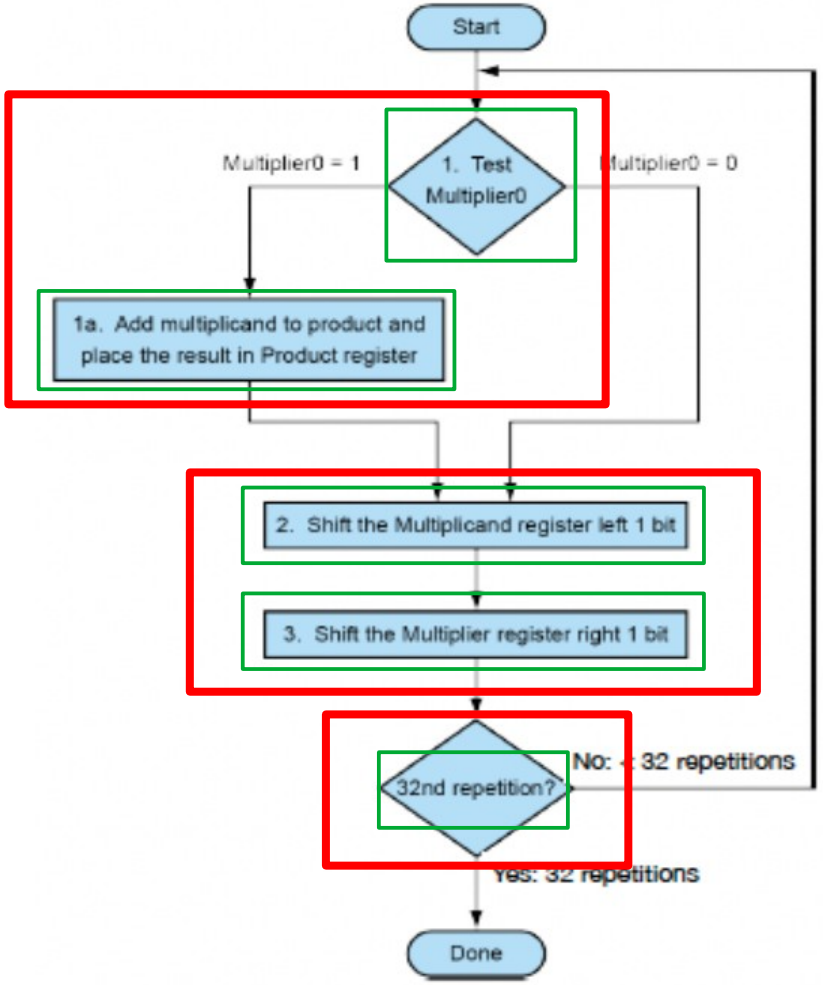
Answer For Question #7

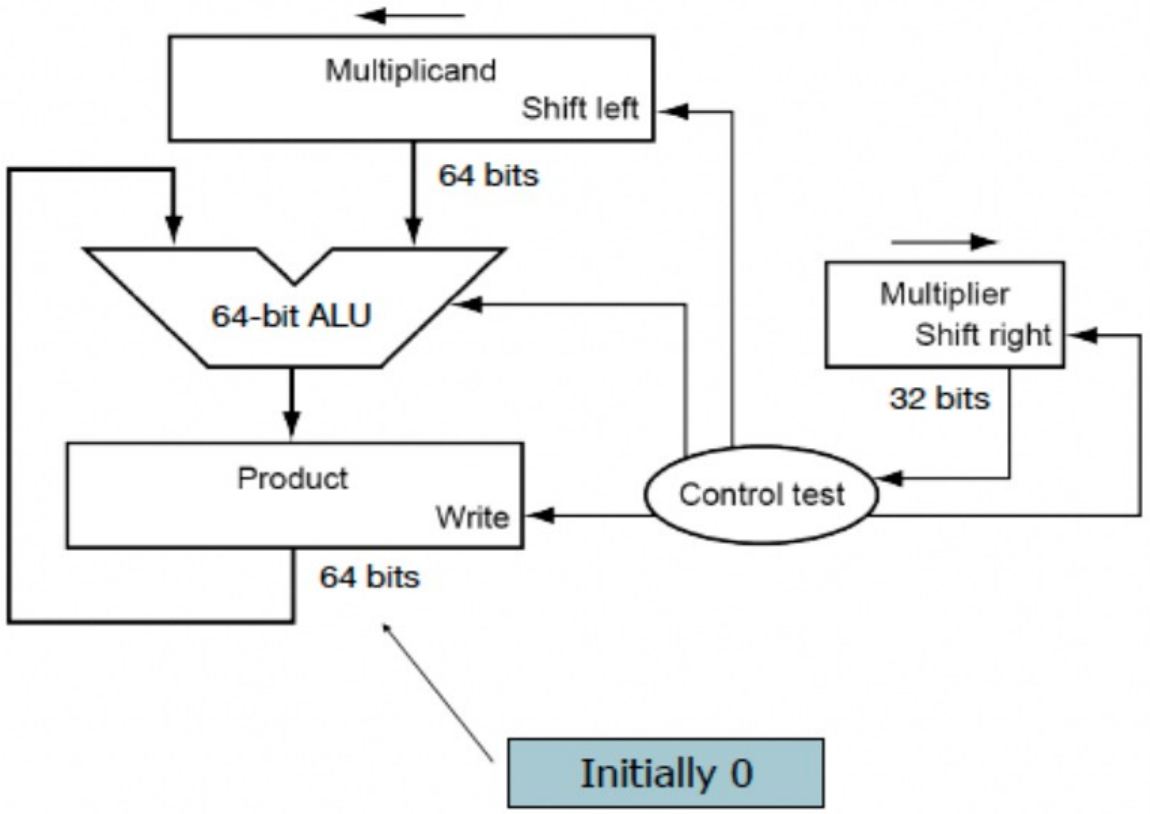
- **Precursor:**
  - General
    - Integer is 8-bits
    - Each step takes 4 time units
    - ‘‘Assume that in step 1a an addition is always performed’’
    - ‘‘Assume that the registers have already been initialized’’
  - Hardware:
    - ‘‘If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously’’
  - Software:
    - ‘‘If this is being done in software, they will have to be done one after the other’’

Answer For Question #7

- **Answer:**
  - Hardware:
    - 1 cycle for addition
    - 1 cycle for *both* shifts
    - 1 cycle to check if we are done
    - **1** + **1** + **1** = 3 cycles in total
  - Software:
    - 1 cycle to decide what to add (i.e. 1 vs. 0)
    - 1 cycle for addition
    - 1 cycle to shift the multiplicand register
    - 1 cycle to shift the multiplier register
    - 1 cycle to check if we are done
    - **1** + **1** + **1** + **1** + **1** = 5 cycles in total

Answer For Question #7





*** Instructions Carried Out By Hardware (Per Cycle)**

*** Instructions Done By Software (Per Cycle)**

Answer For Question #7

- **Answer:**
  - Since the integer is 8 bits wide, each bit has to go through the loop, and each step of the operation takes 4 time units
    - Hardware:
      - 3 cycles x 8 bits x 4 time units = **96 time units**
    - Software:
      - 5 cycles x 8 bits x 4 time units = **160 time units**

Hardware VS. Software

- **Why is this important?**
  - The main idea is that:
    - Things implemented in software can also be done in hardware
    - Things implemented in hardware yield better performance (more efficient) than software
      - However, not everything is possible to do in hardware
      - Making changes to hardware is virtually impossible
        - Reflashing is possible but tedious
    - Things implemented in software can be easily changed, but it is less ‘‘reliable’’

Tutorial Question #8

- **Question:** Calculate the time necessary to perform a multiply using the approach given in Figure below if an integer is 8 bits wide and an adder takes four time units.
- **Answer:** (Homework)
  - *Hints:*
    - Remember that the integer is 8 bits wide
    - Each adder takes 4 time units
    - An adder handles 2 bits
    - Determine how many levels of adders are required
      - Remember that adders on the *same level* execute simultaneously
        - This is key!

Tutorial Answer #8

- **Question:** Calculate the time necessary to perform a multiply using the approach given in Figure below if an integer is 8 bits wide and an adder takes four time units.
- **Answer:** (Hidden)
  - *The adders are arranged in an inverted tree structure. After every level, the number of adders is cut in half*
    - *Since we have 8 bits, we need to use 4 adders in the first level, 2 adders in the second level, and 1 adder in the last level. In total, we need to use 7 adders.*
      - *Since each adder, rather each level takes 4 time units, and we only go through 3 levels, the time it takes is: 3 levels x 4 time units = 12 time units*

Tutorial Answer #8

- **Question:** Calculate the time necessary to perform a multiply using the approach given in Figure below if an integer is 8 bits wide and an adder takes four time units.
- **Answer:** (Hidden)
  - *Another way to compute the number of levels is:*

    $\log_2(A)$, whwere A is the number of adders
    - *So $\log_2(8) = 3$*
      - *Therefore, 3 levels*

THE END