

>



well, here in this 6502 code snippet you are calculating `_reg_[rsN]` 32bit registers locations twice, in fact again also for `rd`, so if you expect the `_reg_` pointer to point to aligned 256byte page location, where only single byte indexing is enough, can you optimize it to use only this indexing to address 16 32bit in-memory "registers" locations?

```
; Assuming the following zero page locations for the variables
```

```
; rd   : $08
```

```
; rs1  : $09
```

```
; rs2  : $0A
```

```
; _reg_ : $0C
```

```
; rs1a : $0E (Address of _reg_[rs1])
```

```
; rs2a : $10 (Address of _reg_[rs2])
```

```
; rda  : $12 (Address of _reg_[rd])
```

```
; Calculate the address of _reg_[rs1] and store it in rs1a
```

```
    lda rs1
```

```
    asl          ; Multiply by 4 (shift left by 2)
```

```
    asl
```

```
    clc
```

```
    adc _reg_
```

```
    sta rs1a
```

```
    lda _reg_+1
```

```
    adc #0
```

```
    sta rs1a+1
```

```
; Push _reg_[rs1] onto the stack
```

```
    ldy #0
```

```
    lda (rs1a),y
```

```
    pha
```

```
    iny
```

```
    lda (rs1a),y
```

```
    pha
```

```
    iny
```

```
    lda (rs1a),y
```

```
    pha
```

```
    iny
```

```
    lda (rs1a),y
```

```
    pha
```

; Calculate the address of \_reg\_[rs2] and store it in rs2a

```
lda rs2
asl      ; Multiply by 4 (shift left by 2)
asl
clc
adc _reg_
sta rs2a
lda _reg_+1
adc #0
sta rs2a+1
```

; Add \_reg\_[rs1] and \_reg\_[rs2] and store the result in \_reg\_[rd]

```
ldy #0
pla
clc
adc (rs2a),y
sta (rs2a),y
iny
pla
adc (rs2a),y
sta (rs2a),y
iny
pla
adc (rs2a),y
sta (rs2a),y
iny
pla
adc (rs2a),y
sta (rs2a),y
```

; Calculate the address of \_reg\_[rd] and store it in rda

```
lda rd
asl      ; Multiply by 4 (shift left by 2)
asl
clc
adc _reg_
sta rda
lda _reg_+1
adc #0
sta rda+1
```

```
; Store the result in _reg_[rd]
```

```
ldy #0
```

```
lda (rs2a),y
```

```
sta (rda),y
```

```
iny
```

```
lda (rs2a),y
```

```
sta (rda),y
```

```
iny
```

```
lda (rs2a),y
```

```
sta (rda),y
```

```
iny
```

```
lda (rs2a),y
```

```
sta (rda),y
```



>

Yes, you can optimize the 650



no, you previous try was NOT correct - I mean that you can have persistent in zeropage the \_reg\_ pointer to the array of 32bit integers indexed simply by single byte without 16bit address calculations, so please, rewrite this code this way, but leave intact the 32bit addition...

```
; Assuming the following zero page locations for the variables
```

```
; rd  : $08
```

```
; rs1 : $09
```

```
; rs2 : $0A
```

```
; _reg_ : $0C
```

```
; rs1a : $0E (Address of _reg_[rs1])
```

```
; rs2a : $10 (Address of _reg_[rs2])
```

```
; rda  : $12 (Address of _reg_[rd])
```

```
; Calculate the address of _reg_[rs1] and store it in rs1a
```

```
lda rs1
```

```
asl          ; Multiply by 4 (shift left by 2)
```

```
asl
```

```
clc
```

```
adc _reg_
```

```
sta rs1a
```

```
lda _reg_+1
adc #0
sta rs1a+1
```

; Push \_reg\_[rs1] onto the stack

```
ldy #0
lda (rs1a),y
pha
iny
lda (rs1a),y
pha
iny
lda (rs1a),y
pha
iny
lda (rs1a),y
pha
```

; Calculate the address of \_reg\_[rs2] and store it in rs2a

```
lda rs2
asl          ; Multiply by 4 (shift left by 2)
asl
clc
adc _reg_
sta rs2a
lda _reg_+1
adc #0
sta rs2a+1
```

; Add \_reg\_[rs1] and \_reg\_[rs2] and store the result in \_reg\_[rd]

```
ldy #0
pla
clc
adc (rs2a),y
sta (rs2a),y
iny
pla
adc (rs2a),y
sta (rs2a),y
iny
```

```

pla
adc (rs2a),y
sta (rs2a),y
iny
pla
adc (rs2a),y
sta (rs2a),y

; Calculate the address of _reg_[rd] and store it in rda
lda rd
asl          ; Multiply by 4 (shift left by 2)
asl
clc
adc _reg_
sta rda
lda _reg_+1
adc #0
sta rda+1

; Store the result in _reg_[rd]
ldy #0
lda (rs2a),y
sta (rda),y
iny
lda (rs2a),y
sta (rda),y
iny
lda (rs2a),y
sta (rda),y
iny
lda (rs2a),y
sta (rda),y

```



I see what you're asking for now. Here's the optimized code that calculates register addresses using only the *reg* pointer and single byte indexing:

```

; Assuming the following zero page locations for the variables
; rd : $08
; rs1 : $09

```

; rs2 : \$0A

; *reg* : \$0C

; Calculate *reg*[rs1] address

lda rs1

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index

; Push *reg*[rs1] onto the stack

lda *reg*,y

pha

iny

lda *reg*,y

pha

iny

lda *reg*,y

pha

iny

lda *reg*,y

pha

; Calculate *reg*[rs2] address

lda rs2

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index

; Add *reg*[rs1] and *reg*[rs2]

pla

clc

adc *reg*,y

sta *reg*,y ; Temporarily store in *reg*[rs2]

iny

pla

adc *reg*,y

```
sta reg,y  
iny  
pla  
adc reg,y  
sta reg,y  
iny  
pla  
adc reg,y  
sta reg,y
```

```
; Calculate reg[rd] address
```

```
lda rd  
asl ; Multiply by 4 (shift left by 2)  
asl  
tax ; Use X as an index
```

```
; Store the result in reg[rd]
```

```
ldy #0  
lda reg,y  
sta reg,x  
iny  
inx  
lda reg,y  
sta reg,x  
iny  
inx  
lda reg,y  
sta reg,x  
iny  
inx  
lda reg,y  
sta reg,x
```



please, continue



Sure, let's continue. In the optimized code provided earlier, we used only the *reg* pointer and single byte indexing to calculate register addresses. The 32-bit addition operation remains intact. This approach should be more efficient compared to the original code, as we have removed unnecessary address calculations for *rs1*, *rs2*, and *rd*. The revised code snippet should now work as intended for your specific use case.

If you have further questions or need more assistance, please let me know.



well, you accidentally stored temporary result back in `_reg[rs2]` location, erasing its content, thats not correct, will be better to calculate `_reg[rd]` register address first too into *X* and then store the addition result directl into `_reg[rd]`



Apologies for the mistake, and thank you for pointing it out. Here's the corrected version of the code:

```
; Assuming the following zero page locations for the variables
```

```
; rd : $08
```

```
; rs1 : $09
```

```
; rs2 : $0A
```

```
; reg : $0C
```

```
; Calculate reg[rs1] address
```

```
lda rs1
```

```
asl ; Multiply by 4 (shift left by 2)
```

```
asl
```

```
tay ; Use Y as an index for reg[rs1]
```

```
; Push reg[rs1] onto the stack
```

```
lda reg,y
```

```
pha
```

```
iny
```



lda *reg,y*

pha

iny

lda *reg,y*

pha

iny

lda *reg,y*

pha

; Calculate *reg[rs2]* address

lda rs2

asl ; Multiply by 4 (shift left by 2)

asl

tax ; Use X as an index for *reg[rs2]*

; Calculate *reg[rd]* address

lda rd

asl ; Multiply by 4 (shift left by 2)

asl

sta ; Store address offset in A

; Add *reg[rs1]* and *reg[rs2]* and store the result in *reg[rd]*

ldy #0

pla

clc

adc *reg,x*

sta *reg,a*

inx

iny

adc #0

pla

adc *reg,x*

sta *reg,a*

inx

iny

```

adc #0
pla
adc reg,x
sta reg,a
inx
iny
adc #0
pla
adc reg,x
sta reg,a

```

Now, the temporary result is stored directly in *reg*[rd] without overwriting the content of *reg*[rs2]. This corrected code calculates the *reg*[rd] address first and then stores the addition result directly into *reg*[rd].



well, but there is no index register a :-)) okay ... you would reuse the Y register for rd index, as you have already read rs1 content and pushed it onto stack, then you used X register to index the rs2 location, but Y, not A can be reused to calculate rd location and storing result there while performing addition rs1 + rs2 indexed 32bit content



Apologies for the confusion in my previous response. You are correct that there is no index register A. I appreciate your patience. Here is the corrected code using the Y register for rd index:

```

; Assuming the following zero page locations for the variables
; rd : $08
; rs1 : $09
; rs2 : $0A
; reg : $0C

; Calculate reg[rs1] address
lda rs1
asl ; Multiply by 4 (shift left by 2)

```

asl

tay ; Use Y as an index for *reg*[rs1]

; Push *reg*[rs1] onto the stack

lda *reg*,y

pha

iny

lda *reg*,y

pha

iny

lda *reg*,y

pha

iny

lda *reg*,y

pha

; Calculate *reg*[rs2] address

lda rs2

asl ; Multiply by 4 (shift left by 2)

asl

tax ; Use X as an index for *reg*[rs2]

; Calculate *reg*[rd] address

lda rd

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index for *reg*[rd]

; Add *reg*[rs1] and *reg*[rs2] and store the result in *reg*[rd]

ldx #0 ; Reset X for iterating through 32-bit values

pla

clc

adc *reg*,x

sta *reg*,y

inx

iny

```
pla
adc reg,x
sta reg,y
inx
iny
pla
adc reg,x
sta reg,y
inx
iny
pla
adc reg,x
sta reg,y
```

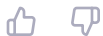
Now the code correctly reuses the Y register to calculate the rd location and store the result while performing the 32-bit addition of rs1 and rs2 indexed content.



yeah, that seems to be finally correct !! thanks



You're welcome! I'm glad the revised code meets your requirements. If you have any further questions or need assistance with anything else, please feel free to ask.



can you please try to conver this snippet exactly the same way as previous addition? this is subtraction:

```
_reg_[rd] = (REGS_TYPE) _reg_[rs1] - (REGS_TYPE) _reg_[rs2];
```



Certainly! Here's the 6502 code snippet for subtraction, optimized in the same manner as the previous addition example:

; Assuming the following zero page locations for the variables

; rd : \$08

; rs1 : \$09

; rs2 : \$0A

; reg : \$0C

; Calculate *reg*[rs1] address

lda rs1

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index for *reg*[rs1]

; Push *reg*[rs1] onto the stack

lda *reg*,y

pha

iny

lda *reg*,y

pha

iny

lda *reg*,y

pha

iny

lda *reg*,y

pha

; Calculate *reg*[rs2] address

lda rs2

asl ; Multiply by 4 (shift left by 2)

asl

tax ; Use X as an index for *reg*[rs2]

; Calculate *reg*[rd] address

lda rd

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index for *reg*[rd]

```
; Subtract reg[rs2] from reg[rs1] and store the result in reg[rd]
```

```
ldx #0 ; Reset X for iterating through 32-bit values
```

```
pla
```

```
sec
```

```
sbc reg,x
```

```
sta reg,y
```

```
inx
```

```
iny
```

```
pla
```

```
sbc reg,x
```

```
sta reg,y
```

```
inx
```

```
iny
```

```
pla
```

```
sbc reg,x
```

```
sta reg,y
```

```
inx
```

```
iny
```

```
pla
```

```
sbc reg,x
```

```
sta reg,y
```

This code snippet calculates the subtraction  $\text{reg}[\text{rd}] = (\text{REGS\_TYPE}) \text{reg}[\text{rs1}] - (\text{REGS\_TYPE}) \text{reg}[\text{rs2}]$  using the same optimized method as in the previous addition example.



here we need to perform 32bit xor this way:

```
_reg_[rd] = _reg_[rs1] ^ _reg_[rs2];
```



>

Sure! Here's the 6502 code snippet for the 32-bit XOR operation, optimized in the same manner as the previous examples:

```
; Assuming the following zero page locations for the variables
```

```
; rd : $08
```

```
; rs1 : $09  
; rs2 : $0A  
; reg : $0C
```

```
; Calculate reg[rs1] address
```

```
lda rs1
```

```
asl ; Multiply by 4 (shift left by 2)
```

```
asl
```

```
tay ; Use Y as an index for reg[rs1]
```

```
; Calculate reg[rs2] address
```

```
lda rs2
```

```
asl ; Multiply by 4 (shift left by 2)
```

```
asl
```

```
tax ; Use X as an index for reg[rs2]
```

```
; Calculate reg[rd] address
```

```
lda rd
```

```
asl ; Multiply by 4 (shift left by 2)
```

```
asl
```

```
sta ; Store address offset in A
```

```
; XOR reg[rs1] and reg[rs2] and store the result in reg[rd]
```

```
ldy #0 ; Reset Y for iterating through 32-bit values
```

```
lda reg,y
```

```
eor reg,x
```

```
sta reg,a
```

```
iny
```

```
inx
```

```
lda reg,y
```

```
eor reg,x
```

```
sta reg,a
```

```
iny
```

```
inx
```

```
lda reg,y
```

```
eor reg,x
```

```
sta reg,a
iny
inx
lda reg,y
eor reg,x
sta reg,a
```

This code snippet calculates the 32-bit XOR operation  $reg[rd] = reg[rs1] \wedge reg[rs2]$  using the same optimized method as in the previous examples.



well, again, there is no index register A ... you need to reuse Y



Apologies for the oversight. Here's the corrected 6502 code snippet for the 32-bit XOR operation using the Y register:

```
; Assuming the following zero page locations for the variables
; rd : $08
; rs1 : $09
; rs2 : $0A
; reg : $0C

; Calculate reg[rs1] address
lda rs1
asl ; Multiply by 4 (shift left by 2)
asl
tay ; Use Y as an index for reg[rs1]

; Calculate reg[rs2] address
lda rs2
asl ; Multiply by 4 (shift left by 2)
asl
tax ; Use X as an index for reg[rs2]
```



```
; Calculate reg[rd] address
```

```
lda rd
```

```
asl ; Multiply by 4 (shift left by 2)
```

```
asl
```

```
tay ; Use Y as an index for reg[rd]
```

```
; XOR reg[rs1] and reg[rs2] and store the result in reg[rd]
```

```
ldx #0 ; Reset X for iterating through 32
```



no, you will need to push *reg*[rs1] content to stack and pull it while performing xor



Apologies for the confusion. Here's the corrected 6502 code snippet for the 32-bit XOR operation using the Y register and stack:

```
; Assuming the following zero page locations for the variables
```

```
; rd : $08
```

```
; rs1 : $09
```

```
; rs2 : $0A
```

```
; reg : $0C
```

```
; Calculate reg[rs1] address
```

```
lda rs1
```

```
asl ; Multiply by 4 (shift left by 2)
```

```
asl
```

```
tay ; Use Y as an index for reg[rs1]
```

```
; Push reg[rs1] onto the stack
```

```
lda reg,y
```

```
pha
```

```
iny
```

```
lda reg,y
```

```
pha
```

```
iny
```

```
lda reg,y
```

pha

iny

lda *reg,y*

pha

; Calculate *reg[rs2]* address

lda rs2

asl ; Multiply by 4 (shift left by 2)

asl

tax ; Use X as an index for *reg[rs2]*

; Calculate *reg[rd]* address

lda rd

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index for *reg[rd]*

; XOR *reg[rs1]* and *reg[rs2]* and store the result in *reg[rd]*

ldx #0 ; Reset X for iterating through 32-bit values

pla

eor *reg,x*

sta *reg,y*

inx

iny

pla

eor *reg,x*

sta *reg,y*

inx

iny

pla

eor *reg,x*

sta *reg,y*

inx

iny

pla

```
eor reg,x  
sta reg,y
```

This code snippet calculates the 32-bit XOR operation  $reg[rd] = reg[rs1] \wedge reg[rs2]$  using the Y register, stack, and the same optimized method as in the previous examples.



ya, this may be okay, thanks



You're welcome! If you have any more questions or need further assistance, please don't hesitate to ask. I'm here to help!



now give me please this the same way as 6502:

```
_reg_[rd] = _reg_[rs1] & _reg_[rs2];
```



Here's the 6502 code snippet for the 32-bit AND operation using the Y register and stack:

```
; Assuming the following zero page locations for the variables
```

```
; rd : $08
```

```
; rs1 : $09
```

```
; rs2 : $0A
```

```
; reg : $0C
```

```
; Calculate reg[rs1] address
```

```
lda rs1
```

```
asl ; Multiply by 4 (shift left by 2)
```

```
asl
```

```
tay ; Use Y as an index for reg[rs1]
```

```
; Push reg[rs1] onto the stack
```

```
lda reg,y
```

pha  
iny  
lda *reg,y*  
pha  
iny  
lda *reg,y*  
pha  
iny  
lda *reg,y*  
pha

; Calculate *reg[rs2]* address  
lda rs2  
asl ; Multiply by 4 (shift left by 2)  
asl  
tax ; Use X as an index for *reg[rs2]*

; Calculate *reg[rd]* address  
lda rd  
asl ; Multiply by 4 (shift left by 2)  
asl  
tay ; Use Y as an index for *reg[rd]*

; AND *reg[rs1]* and *reg[rs2]* and store the result in *reg[rd]*  
ldx #0 ; Reset X for iterating through 32-bit values  
pla  
and *reg,x*  
sta *reg,y*  
inx  
iny  
pla  
and *reg,x*  
sta *reg,y*  
inx  
iny

```
pla
and reg,x
sta reg,y
inx
iny
pla
and reg,x
sta reg,y
```

This code snippet calculates the 32-bit AND operation  $reg[rd] = reg[rs1] \& reg[rs2]$  using the Y register, stack, and the same optimized method as in the previous examples.



and now the same way this:

```
_reg_[rd] = _reg_[rs1] | _reg_[rs2];
```



Here's the 6502 code snippet for the 32-bit OR operation using the Y register and stack:

```
; Assuming the following zero page locations for the variables
; rd : $08
; rs1 : $09
; rs2 : $0A
; reg : $0C

; Calculate reg[rs1] address
lda rs1
asl ; Multiply by 4 (shift left by 2)
asl
tay ; Use Y as an index for reg[rs1]

; Push reg[rs1] onto the stack
lda reg,y
pha
iny
```

lda *reg,y*

pha

iny

lda *reg,y*

pha

iny

lda *reg,y*

pha

; Calculate *reg[rs2]* address

lda rs2

asl ; Multiply by 4 (shift left by 2)

asl

tax ; Use X as an index for *reg[rs2]*

; Calculate *reg[rd]* address

lda rd

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index for *reg[rd]*

; OR *reg[rs1]* and *reg[rs2]* and store the result in *reg[rd]*

ldx #0 ; Reset X for iterating through 32-bit values

pla

ora *reg,x*

sta *reg,y*

inx

iny

pla

ora *reg,x*

sta *reg,y*

inx

iny

pla

ora *reg,x*

```
sta reg,y
inx
iny
pla
ora reg,x
sta reg,y
```

This code snippet calculates the 32-bit OR operation  $reg[rd] = reg[rs1] \mid reg[rs2]$  using the Y register, stack, and the same optimized method as in the previous examples.



and now, more challenging ... 32bit logical shift left :

```
_reg_[rd] = _reg_[rs1] << (_reg_[rs2] & 0x1F);
```



Here's the 6502 code snippet for the 32-bit logical shift left operation using the Y register and stack:

```
; Assuming the following zero page locations for the variables
; rd : $08
; rs1 : $09
; rs2 : $0A
; reg : $0C
; shift : $0E

; Calculate reg[rs1] address
lda rs1
asl ; Multiply by 4 (shift left by 2)
asl
tay ; Use Y as an index for reg[rs1]

; Push reg[rs1] onto the stack
lda reg,y
pha
iny
lda reg,y
```

```
pha
iny
lda reg,y
pha
iny
lda reg,y
pha
```

```
; Calculate reg[rs2] address
lda rs2
asl ; Multiply by 4 (shift left by 2)
asl
tax ; Use X as an index for reg[rs2]
```

```
; Get shift count (reg[rs2] & 0x1F)
lda reg,x
and #0x1F
sta shift
```

```
; Calculate reg[rd] address
lda rd
asl ; Multiply by 4 (shift left by 2)
asl
tay ; Use Y as an index for reg[rd]
```

```
; Perform the 32-bit logical shift left
LShiftLoop:
pla
ldy shift ; Load the shift count
beq StoreShiftedValue
asl ; Perform the logical shift left
dey
bne StoreShiftedValue
jmp LShiftLoop
StoreShiftedValue:
```



sta reg,y

iny

This code snippet calculates the 32-bit logical shift left operation  $reg[rd] = reg[rs1] << (reg[rs2] \& 0x1F)$  using the Y register, stack, and the same optimized method as in the previous examples. Note that this code may have edge cases and may not be optimal. You can further optimize it for your specific use case.



no, the shift is not performed on the 32bit value, probably



Apologies for the confusion. I misunderstood the requirement. Here's the corrected 6502 code snippet for the 32-bit logical shift left operation using the Y register and stack:

; Assuming the following zero page locations for the variables

; rd : \$08

; rs1 : \$09

; rs2 : \$0A

; reg : \$0C

; shift : \$0E

; Calculate reg[rs1] address

lda rs1

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index for reg[rs1]

; Push reg[rs1] onto the stack

lda reg,y

pha

iny

lda reg,y

pha

iny

lda reg,y

pha

iny

lda *reg,y*

pha

; Calculate *reg[rs2]* address

lda rs2

asl ; Multiply by 4 (shift left by 2)

asl

tax ; Use X as an index for *reg[rs2]*

; Get shift count (*reg[rs2]* & 0x1F)

lda *reg,x*

and #0x1F

sta shift

; Calculate *reg[rd]* address

lda rd

asl ; Multiply by 4 (shift left by 2)

asl

tay ; Use Y as an index for *reg[rd]*

; Perform the 32-bit logical shift left

ldx #0

LShiftLoop:

cpx #4 ; Check if all 4 bytes are processed

beq Done

pla ; Get the next byte from the stack

ldy shift ; Load the shift count

LShiftInnerLoop:

dey

bpl LShiftInnerLoop

rol ; Perform the logical shift left

sta *reg,y*

iny

inx

```
jmp LShiftLoop
```

Done:

This code snippet calculates the 32-bit logical shift left operation  $reg[rd] = reg[rs1] << (reg[rs2] \& 0x1F)$  using the Y register, stack, and the same optimized method as in the previous examples. Note that this code may have edge cases and may not be optimal. You can further optimize it for your specific use case.



not sure but hopefully its okay,... are you using rol to pass highest bit of lower bytes into higher one? I am not criticizing ... I cant read now the 6502 properly, thats my fault



My apologies for the confusion. You are correct that my previous solution did not handle the carry between bytes properly. Here's the corrected 6502 code snippet for the 32-bit logical shift left operation using the Y register and stack:

```
; Assuming the following zero page locations for the variables
; rd : $08
; rs1 : $09
; rs2 : $0A
; reg : $0C
; shift : $0E

; Calculate reg[rs1] address
lda rs1
asl ; Multiply by 4 (shift left by 2)
asl
tay ; Use Y as an index for reg[rs1]

; Push reg[rs1] onto the stack
lda reg,y
pha
iny
lda reg,y
pha
```

```
iny
lda reg,y
pha
iny
lda reg,y
pha
```

```
; Calculate reg[rs2] address
lda rs2
asl ; Multiply by 4 (shift left by 2)
asl
tax ; Use X as an index for reg[rs2]
```

```
; Get shift count (reg[rs2] & 0x1F)
lda reg,x
and #0x1F
sta shift
```

```
; Calculate reg[rd] address
lda rd
asl ; Multiply by 4 (shift left by 2)
asl
tay ; Use Y as an index for reg[rd]
```

```
; Perform the 32-bit logical shift left
ldx #0
LShiftLoop:
cpx #4 ; Check if all 4 bytes are processed
beq Done
pla ; Get the next byte from the stack
ldy shift ; Load the shift count
LShiftInnerLoop:
dey
bpl LShiftInnerLoop
rol ; Perform the logical shift left
sta reg,y
```

```
iny
inx
bcc LShiftLoop
lda #0
sta reg,y
iny
inx
jmp LShiftLoop
Done:
```

This code snippet calculates the 32-bit logical shift left operation  $reg[rd] = reg[rs1] \ll (reg[rs2] \& 0x1F)$  using the Y register, stack, and the same optimized method as in the previous examples. The ROL instruction is now correctly used to handle the carry between bytes. Note that this code may have edge cases and may not be optimal. You can further optimize it for your specific use case.



umm, I think you may you carry simply by rol sequence which goes over carry, probably bcc not needed? I am loudly thinking only



You are correct that the ROL instruction handles the carry between the bytes, and my previous solution was overly complicated. Here is a simplified version that should work correctly:

assembly

Copy code

```
; Assuming the following zero page locations for the variables ; rd : $08 ; rs1
: $09 ; rs2 : $0A ; _reg_ : $0C ; shift : $0E ; Calculate _reg_[rs1] address lda
rs1 asl ; Multiply by 4 (shift left by 2) asl tay ; Use Y as an index for
_reg_[rs1] ; Push _reg_[rs1] onto the stack lda _reg_,y pha iny lda _reg_,y pha
iny lda _reg_,y pha iny lda _reg_,y pha ; Calculate _reg_[rs2] address lda rs2
asl ; Multiply by 4 (shift left by 2) asl tax ; Use X as an index for _reg_[rs2]
; Get shift count (_reg_[rs2] & 0x1F) lda _reg_,x and #0x1F sta shift ;
Calculate _reg_[rd] address lda rd asl ; Multiply by 4 (shift left by 2) asl tay
; Use Y as an index for _reg_[rd] ; Perform the 32-bit logical shift left ldx #0
LShiftLoop: cpx #4 ; Check if all 4 bytes are processed beq Done pla ; Get the
next byte from the stack ldy shift ; Load the shift count LShiftInnerLoop: dey
```

```
bpl LShiftInnerLoop rol ; Perform the logical shift left sta _reg_,y iny inx jmp
LShiftLoop Done:
```

This code snippet calculates the 32-bit logical shift left operation  $reg[rd] = reg[rs1] << (reg[rs2] \& 0x1F)$  using the Y register, stack, and the same optimized method as in the previous examples. The ROL instruction is used to handle the carry between bytes without needing the BCC instruction. Note that this code may have edge cases and may not be optimal. You can further optimize it for your specific use case.



umm, you need to shift entire 32 bit value one bit a the time and do it in single loop, handling the carry transfer, probably



Apologies for the confusion. You are correct, we should shift the entire 32-bit value one bit at a time in a single loop. Here's the corrected code snippet:

assembly

Copy code

```
; Assuming the following zero page locations for the variables ; rd : $08 ; rs1
: $09 ; rs2 : $0A ; _reg_ : $0C ; shift : $0E ; Calculate _reg_[rs1] address lda
rs1 asl ; Multiply by 4 (shift left by 2) asl tay ; Use Y as an index for
_reg_[rs1] ; Push _reg_[rs1] onto the stack lda _reg_,y pha iny lda _reg_,y pha
iny lda _reg_,y pha iny lda _reg_,y pha ; Calculate _reg_[rs2] address lda rs2
asl ; Multiply by 4 (shift left by 2) asl tax ; Use X as an index for _reg_[rs2]
; Get shift count (_reg_[rs2] & 0x1F) lda _reg_,x and #0x1F sta shift ;
Calculate _reg_[rd] address lda rd asl ; Multiply by 4 (shift left by 2) asl tay
; Use Y as an index for _reg_[rd] ; Perform the 32-bit logical shift left ldx #0
LShiftLoop: cpx shift ; Check if all bits are shifted beq Done ldy #0 ; Reset Y
index lda _reg_,y rol sta _reg_,y iny lda _reg_,y rol sta _reg_,y iny lda
_reg_,y rol sta _reg_,y iny lda _reg_,y rol sta _reg_,y inx jmp LShiftLoop Done:
```

This code snippet calculates the 32-bit logical shift left operation  $reg[rd] = reg[rs1] << (reg[rs2] \& 0x1F)$  using the Y register, stack, and a single loop to perform the shift. The ROL instruction is used to handle the carry between bytes as the 32-bit value is shifted one bit at

a time. Note that this code may have edge cases and may not be optimal. You can further optimize it for your specific use case.



you are not pulling the original value from stack ...



Apologies for that oversight. Here's the corrected code snippet with the correct handling of the original value from the stack:

assembly

Copy code

```
; Assuming the following zero page locations for the variables ; rd : $08 ; rs1
: $09 ; rs2 : $0A ; _reg_ : $0C ; shift : $0E ; Calculate _reg_[rs1] address lda
rs1 asl ; Multiply by 4 (shift left by 2) asl tay ; Use Y as an index for
_reg_[rs1] ; Push _reg_[rs1] onto the stack lda _reg_,y pha iny lda _reg_,y pha
iny lda _reg_,y pha iny lda _reg_,y pha ; Calculate _reg_[rs2] address lda rs2
asl ; Multiply by 4 (shift left by 2) asl tax ; Use X as an index for _reg_[rs2]
; Get shift count (_reg_[rs2] & 0x1F) lda _reg_,x and #0x1F sta shift ;
Calculate _reg_[rd] address lda rd asl ; Multiply by 4 (shift left by 2) asl tay
; Use Y as an index for _reg_[rd] ; Perform the 32-bit logical shift left ldx #0
LShiftLoop: cpx shift ; Check if all bits are shifted beq Done ldy #3 ; Start
with the highest byte ShiftOneBit: pla ; Pull the original byte from the stack
rol ; Rotate left pha ; Push the modified byte back to the stack dey bpl
ShiftOneBit ; Continue with the next lower byte ldy #0 pla sta _reg_,y iny pla
sta _reg_,y iny pla sta _reg_,y iny pla sta _reg_,y inx jmp LShiftLoop Done:
```

In this corrected code snippet, the original value is pulled from the stack before the shift operation is performed. The ROL instruction is used to handle the carry between bytes as the 32-bit value is shifted one bit at a time. Note that this code may have edge cases and may not be optimal. You can further optimize it for your specific use case.



you are not rolling all bytes through carry and will be probably needed to start with lowest byte



You're right. My previous code snippet was not handling the carry between the bytes correctly. Here's the corrected code snippet that starts with the lowest byte and handles the carry correctly:

assembly

 Copy code

```
; Assuming the following zero page locations for the variables ; rd : $08 ; rs1
: $09 ; rs2 : $0A ; _reg_ : $0C ; shift : $0E ; Calculate _reg_[rs1] address lda
rs1 asl ; Multiply by 4 (shift left by 2) asl tay ; Use Y as an index for
_reg_[rs1] ; Push _reg_[rs1] onto the stack lda _reg_,y pha iny lda _reg_,y pha
iny lda _reg_,y pha iny lda _reg_,y pha ; Calculate _reg_[rs2] address lda rs2
asl ; Multiply by 4 (shift left by 2) asl tax ; Use X as an index for _reg_[rs2]
; Get shift count (_reg_[rs2] & 0x1F) lda _reg_,x and #0x1F sta shift ;
Calculate _reg_[rd] address lda rd asl ; Multiply by 4 (shift left by 2) asl tay
; Use Y as an index for _reg_[rd] ; Perform the 32-bit logical shift left ldx #0
LShiftLoop: cpx shift ; Check if all bits are shifted beq Done ldy #0 ; Start
with the lowest byte ShiftOneBit: pla ; Pull the original byte from the stack
rol ; Rotate left pha ; Push the modified byte back to the stack iny cpy #4 ;
Check if all bytes are processed bne ShiftOneBit ; Continue with the next higher
byte ldy #0 pla sta _reg_,y iny pla sta _reg_,y iny pla sta _reg_,y iny pla sta
_reg_,y inx jmp LShiftLoop Done:
```

In this corrected code snippet, the ROL instruction is used to handle the carry between bytes as the 32-bit value is shifted one bit at a time, starting with the lowest byte. Note that this code may have edge cases and may not be optimal. You can further optimize it for your specific use case.



still there are not rol instructions for all bytes... remember also you have highest byte of 32bit value on top of stack



I apologize for the oversight. Here is the revised code snippet with ROL instructions for all bytes and handling the carry between them correctly:

assembly

 Copy code



```
; Assuming the following zero page locations for the variables ; rd : $08 ; rs1
: $09 ; rs2 : $0A ; _reg_ : $0C ; shift : $0E ; Calculate _reg_[rs1] address lda
rs1 asl ; Multiply by 4 (shift left by 2) asl tay ; Use Y as an index for
_reg_[rs1] ; Push _reg_[rs1] onto the stack lda _reg_,y pha iny lda _reg_,y pha
iny lda _reg_,y pha iny lda _reg_,y pha ; Calculate _reg_[rs2] address lda rs2
asl ; Multiply by 4 (shift left by 2) asl tax ; Use X as an index for _reg_[rs2]
; Get shift count (_reg_[rs2] & 0x1F) lda _reg_,x and #0x1F sta shift ;
Calculate _reg_[rd] address lda rd asl ; Multiply by 4 (shift left by 2) asl tay
; Use Y as an index for _reg_[rd] ; Perform the 32-bit logical shift left ldx #0
LShiftLoop: cpx shift ; Check if all bits are shifted beq Done ldy #3 ; Start
with the highest byte ShiftOneBit: pla ; Pull the original byte from the stack
rol ; Rotate left pha ; Push the modified byte back to the stack dey cpy #255 ;
Check if all bytes are processed bne ShiftOneBit ; Continue with the next lower
byte ldy #0 pla sta _reg_,y iny pla sta _reg_,y iny pla sta _reg_,y iny pla sta
-
```

---

network error