Ttpasm Study Guide

1 TTPASM Complete Study Guide for Final Exam

1.1 Table of Contents

1.2 Architecture Fundamentals

1.2.1 Registers

TTPASM has **4 general-purpose registers**, all 1 byte (8 bits) wide:

- **A, B, C**: General purpose (not preserved across function calls)
- **D**: Designated **stack pointer** by convention (MUST be preserved)

1.2.2 Memory Model

- **Byte-addressable**: Each memory location stores 1 byte
- **Word width**: 1 byte (no alignment issues!)
- **Stack grows DOWNWARD** (toward lower addresses)
- Stack pointer (D) **always points to the last item pushed**

1.2.3 Flags

The processor has status flags affected by certain operations:

- **Z (Zero flag)**: Set when result is zero
- **C (Carry flag)**: Set when result causes carry/borrow

1.3 Instruction Set Reference

1.3.1 Data Movement

‘ldi reg, value‘Load Immediate - Load a constant or label address into a register

```
ldi a, 5        // a = 5 (constant)
ldi a, x        // a = &x (address of label x)
ldi a, . 5 +    // a = PC + 5 (relative addressing for return addresses)
```

‘ld reg, (reg)‘Load from Memory - Dereference a pointer

```
ldi a, x        // a = &x
ld a, (a)       // a = *a = x (load value at address)
```

‘st (reg), reg‘Store to Memory - Write to memory

```
ldi a, x        // a = &x
ldi b, 5        // b = 5
st (a), b       // *a = b, i.e., x = 5
```

‘cpr dest, src‘Copy Register - Copy value from one register to another

```
cpr c, d        // c = d (often used to save stack pointer)
```

1.3.2 Arithmetic

‘add dest, src‘Add - dest = dest + src

```
ldi a, 3
ldi b, 5
add a, b        // a = 3 + 5 = 8
```

‘sub dest, src‘Subtract - dest = dest - src

```
ldi a, 10
ldi b, 3
sub a, b        // a = 10 - 3 = 7
```

‘inc reg‘Increment - reg = reg + 1

```
inc d           // d = d + 1
```

‘dec reg‘Decrement - reg = reg - 1

```
dec d           // d = d - 1
```

1.3.3 Logical

‘and dest, src‘Bitwise AND - dest = dest src, **sets Z flag**

```
and a, a          // a = a & a = a, but Z flag is set if a==0
jzi isZero        // jump if a was zero
```

Common use: Testing if a register is zero without modifying it

1.3.4 Comparison

‘cmp reg1, reg2‘Compare - Computes reg1 - reg2, sets flags but **doesn’t store result**

```
ldi a, 5
ldi b, 3
cmp a, b          // computes a-b, sets C and Z flags
jci less          // jump if a < b (carry set)
jzi equal         // jump if a == b (zero set)
```

1.3.5 Control Flow

‘jmpi label‘Jump Immediate - Unconditional jump to label

```
jmpi main         // jump to label 'main'
```

‘jmp reg‘Jump Register - Jump to address in register

```
ldi a, someFunc
jmp a             // jump to address in a
```

‘jzi label‘Jump if Zero - Conditional jump if Z flag is set

```
cmp a, b
jzi equal         // jump if a == b
```

‘jci label‘Jump if Carry - Conditional jump if C flag is set

```
cmp a, b
jci less          // jump if a < b
```

1.3.6 Other

‘nop‘No Operation - Does nothing (used for simulator compatibility)
‘halt‘Halt - Stop execution

1.4 Stack Operations

1.4.1 The Stack Convention

CRITICAL RULES:

1. Stack pointer (D) **always points to the last item pushed**
2. Stack grows **downward** (push decrements D, pop increments D)
3. Items pushed earlier have **higher addresses**
4. Anything **below** where D points can be overwritten (by interrupts, etc.)

1.4.2 Push Operation

```
// To push register X onto stack:
dec d             // Reserve space (move stack pointer down)
st (d), x         // Store value at new stack top
```

C equivalent: `*(--D) = X;`

1.4.3 Pop Operation

```
// To pop from stack into register X:
ld x, (d)         // Load value from stack top
inc d             // Deallocate space (move stack pointer up)
```

C equivalent: `X = *(D++);`

1.4.4 Stack Visualization

After `push 2` then `push 13`:

```
Address  | Content | Offset from D
---------|---------|---------------
...      | ...     |
0x0105   | 2       | D+1
0x0104   | 13      | D+0 ← D points here
0x0103   | ???     | (can be overwritten!)
```

Key insight: `D+0` means ”where D points”, `D+1` means ”one byte higher than D”

1.5 Control Structures

1.5.1 The Translation Process

All C control structures reduce to:

1. **Conditional goto** statements
2. **Labels**
3. **Unconditional goto** statements

1.5.2 If Statement

Without else:

```
if (condition) {
    blk1;
}
```

Translates to:

```
if (!condition) goto endIf;
blk1;
endIf:
```

```
    blk1;
} else {
    blk2;
}
```

Translates to:

```
if (!condition) goto else_label;
blk1;
goto endIf;
else_label:
blk2;
endIf:
```

1.5.3 While Loop (Pre-checking)

```
while (condition) {
    blk1;
}
```

Translates to:

```
beginLoop:
if (!condition) goto endLoop;
blk1;
goto beginLoop;
endLoop:
```

1.5.4 Do-While Loop (Post-checking)

```
do {
    blk1;
} while (condition);
```

Translates to:

```
beginLoop:
blk1;
if (condition) goto beginLoop;
```

1.5.5 Boolean Operators

NOT operator

```
if (!c) goto L1;
```

Transforms to:

```
if (c) goto L2;
goto L1;
L2:
```

OR operator

```
if (c || d) goto L1;
```

Becomes:

```
if (c) goto L1;
if (d) goto L1;
```

AND operator

```
if (c && d) goto L1;
```

Becomes:

```
if (!c) goto L2;
if (d) goto L1;
L2:
```

1.5.6 Comparison Operators

TTPASM only has native support for `<` and `==`. Others must be derived:

C Operator	Transformation
‘x $<$ y‘	Just that (native)
‘x $>$ y‘	‘y $<$ x‘ (reverse)
‘x $<=$ y‘	‘(x $<$ y)
‘x $>=$ y‘	‘(y $<$ x)
‘x $==$ y‘	Just that (native)
‘x != y‘	‘!(x == y)‘

1.5.7 Implementing Comparisons

Testing ‘x < y‘:

```
ldi a, x
ld a, (a)       // a = x
ldi b, y
ld b, (b)       // b = y
cmp a, b        // compute a - b
jci less        // jump if carry (x < y)
```

Testing ‘x == y‘:

```
ldi a, x
ld a, (a)       // a = x
ldi b, y
ld b, (b)       // b = y
cmp a, b        // compute a - b
jzi equal       // jump if zero (x == y)
```

Testing ‘x != y‘:

```
ldi a, x
ld a, (a)       // a = x
```

```
ld b, (b)        // b = y
cmp a, b         // compute a - b
jzi skip         // if equal, skip the jump
jmpi notEqual    // not equal, take this path
skip:
```

Testing ‘x > y‘ (reverse to ‘y < x‘):

```
ldi a, y         // Note: reversed!
ld a, (a)        // a = y
ldi b, x
ld b, (b)        // b = x
cmp a, b         // compute y - x
jci greater      // jump if y < x, i.e., x > y
```

Testing ‘x >= y‘ (becomes ‘(y < x) —— (y == x)‘):

```
ldi a, y
ld a, (a)        // a = y
ldi b, x
ld b, (b)        // b = x
cmp a, b         // compute y - x
jci greaterEqual // jump if y < x (x > y)
jzi greaterEqual // jump if y == x (x == y)
```

1.5.8 Label Naming Conventions

Sequential numbering for same nesting level:

```
if (x >= y) goto endIf1;
x++;
endIf1:
if (y >= x) goto endIf2;
y++;
endIf2:
```

Nested constructs use suffix notation:

```
beginLoop1:
    if (x >= 3) goto endLoop1;
    if (y != x) goto loop1_endIf1;
    z++;
    loop1_endIf1:
    x++;
    goto beginLoop1;
endLoop1:
```

1.6 Function Calls \ Caller-Callee Agreement

1.6.1 The Agreement (CRITICAL!)

This is the **contract** between caller and callee. Violations cause bugs!

1.6.2 Caller Responsibilities

1. **Push arguments in REVERSE order** (last argument first)
2. **Push return address**
3. **Jump to callee**
4. **Clean up arguments** after return
5. **Retrieve return value from register A**

Example: Calling ‘subtract(3, 5)‘

```
// subtract(3, 5);
ldi a, 5
dec d
st (d), a        // push second argument (5)

ldi a, 3
dec d
st (d), a        // push first argument (3)

ldi a, . 5 +     // compute return address (PC + 5 instructions)
dec d
st (d), a        // push return address

jmpi subtract    // jump to function

// Return point (after function returns)
inc d            // deallocate first argument
inc d            // deallocate second argument
// result is now in register A
```

1.6.3 Callee Responsibilities

1. **Allocate local variables** (if any)
2. **Access parameters** at positive offsets from D
3. **Perform function logic**
4. **Place return value in register A** (if returning a value)
5. **Deallocate local variables**
6. **Pop and jump to return address**

Example: Function ‘int subtract(int x, int y)‘

```
subtract:
    // At entry: D points to return address
    // Stack: [retAddr] x y

    // Access parameters
    cpr c, d         // c = copy of stack pointer
    ldi a, 1         // offset to parameter x
    add c, a         // c = address of x
    ld a, (c)        // a = x
    inc c            // c = address of y
```

```
    // Perform operation
    sub a, b        // a = x - y

    // Return (a already has result)
    ld b, (d)       // b = return address
    inc d           // pop return address
    jmp b           // return to caller
```

1.6.4 Call Frame Layout

After caller pushes arguments and return address, before callee allocates locals:

```
Offset from D | Content
--------------|------------------
+2            | last argument (y)
+1            | first argument (x)
+0            | return address ← D points here
```

After callee allocates local variables (e.g., 2 bytes for `a` and `b`):

```
Offset from D | Content
--------------|------------------
+4            | last argument (y)
+3            | first argument (x)
+2            | return address
+1            | local variable b
+0            | local variable a ← D points here
```

1.6.5 Accessing Frame Items

Use **labels to define offsets**:

```
someFunc:
    // Define offsets for local variables
    someFunc_a: 0                            // offset to local var a
    someFunc_b: someFunc_a + 1               // offset to local var b
    someFunc_localVarSize: someFunc_b + 1    // total bytes for locals (2)

    // Define offsets for parameters
    someFunc_x: someFunc_localVarSize + 1    // +1 to skip return address
    someFunc_y: someFunc_x + 1

    // Allocate local variables
    ldi a, someFunc_localVarSize
    sub d, a        // d = d - 2 (allocate 2 bytes)

    // Access a parameter (e.g., x)
    ldi a, someFunc_x    // a = offset to x
    add a, d             // a = address of x
    ld a, (a)            // a = value of x

    // Access a local variable (e.g., b)
    ldi b, someFunc_b    // b = offset to b
    add b, d             // b = address of b
    // Now can load/store to (b)

    // ... function body ...

    // Deallocate local variables
    ldi b, someFunc_localVarSize
    add d, b        // d = d + 2

    // Return
    ld b, (d)       // b = return address
    inc d           // pop return address
    jmp b
```

1.6.6 Important Rules

1. **Never modify D directly** except for:

- Allocating/deallocating locals - Pushing/popping stack items

1. **Registers A, B, C are NOT preserved** across calls

- If you need their values after a call, save them first!

1. **D must be preserved** - callee must restore D to point to return address before returning

1. **Return values go in register A**

1. **Caller cleans up arguments**, not callee!

1.7 Arrays and Structures

1.7.1 Arrays

Declaration

```
uint8_t buffer[10];  // 10 bytes
```

Total size: `sizeof(element)` *`length`* `= 1  10 = 10 bytes`

Indexing Formula

`&buffer[i] = &buffer + (sizeof(element) * i)`

Accessing Array Elements (1-byte elements)

```
// Assume: c = &buffer, b = index i
// Goal: access buffer[i]

add b, c        // b = &buffer + i = &buffer[i]
```

Accessing Array Elements (multi-byte elements) For 4-byte elements:

```
// Assume: c = &buffer, b = index i
// Goal: access buffer[i] where each element is 4 bytes

add b, b        // b = 2 * i
add b, b        // b = 4 * i
add b, c        // b = &buffer + (4 * i)
```

Limitation: TTPASM has no multiplication, so only powers of 2 are practical!

Pointer Arithmetic Incrementing a pointer:

```
ptr++;  // In C, this adds sizeof(*ptr) to ptr
```

```
// Assume: a = ptr, and elements are TYPEX_size bytes
ldi b, TYPEX_size
add a, b        // a = ptr + 1 (in pointer arithmetic)
```

Sequential Access Pattern Instead of indexing, use pointer increment:

```
// Original (using indexing)
for (i = 0, sum = 0; i < N; ++i) {
    sum += a[i];
}

// Better for TTPASM (using pointer)
for (i = 0, sum = 0, ptr = a; i < N; ++i) {
    sum += *(ptr++);
}
```

1.7.2 Structures

C Definition

```
struct X {
    uint8_t x;      // 1 byte
    struct X *ptr;  // 1 byte (pointer)
    uint8_t y;      // 1 byte
};
```

TTPASM Definition (using labels)

```
X_x: 0              // offset to member x (0 bytes from start)
X_ptr: X_x + 1      // offset to member ptr (1 byte from start)
X_y: X_ptr + 1      // offset to member y (2 bytes from start)
X_size: X_y + 1     // total size of struct (3 bytes)
```

Accessing Structure Members

```
// Assume: b = address of a struct X instance
// Goal: access member y

ldi a, X_y      // a = offset to member y (2)
add b, a        // b = address of struct + offset = &(struct.y)
ld a, (b)       // a = struct.y
```

No Alignment Issues! Because TTPASM has a word width of 1 byte, there are **no alignment or padding issues**. Members are always contiguous!

1.8 Common Patterns \ Idioms

1.8.1 Pattern 1: Loading a Global Variable

```
// To load global variable x into register a:
ldi a, x        // a = &x
ld a, (a)       // a = *a = x
```

1.8.2 Pattern 2: Storing to a Global Variable

```
// To store value in register b to global variable x:
ldi a, x        // a = &x
st (a), b       // *a = b, i.e., x = b
```

1.8.3 Pattern 3: Testing if Register is Zero

```
// Test if register a is zero without destroying it:
and a, a        // a = a & a = a, but sets Z flag
jzi isZero      // jump if a == 0
```

1.8.4 Pattern 4: Negating a Condition

```
// if (!(x < y)) goto label;
// Becomes: if (x >= y) goto label;
// Which is: if ((y < x) || (y == x)) goto label;

ldi a, y
ld a, (a)
ldi b, x
ld b, (b)
cmp a, b
jci label       // y < x, so x > y, so x >= y
jzi label       // y == x, so x >= y
```

1.8.5 Pattern 5: Saving Stack Pointer

```
// Save current stack pointer to register c:
cpr c, d        // c = d

// Now c can be used to access frame items while d changes
```

1.8.6 Pattern 6: Relative Return Address

```
// Push return address for next instruction after 5 more instructions:
ldi a, . 5 +    // a = PC + 5
dec d
st (d), a       // push return address
```

```
ldi a, . 5 +      // 1: this instruction
dec d             // 2: decrement stack
st (d), a         // 3: store return address
jmpi func         // 4: jump to function
inc d             // 5: this is the return point!     PC + 5
```

1.8.7 Pattern 7: Recursive Function Template

```
func:
    // Define offsets
    func_localVarSize: 0          // adjust if locals exist
    func_param1: func_localVarSize + 1

    // Allocate locals (if any)
    ldi a, func_localVarSize
    sub d, a

    // Base case check
    ldi a, func_param1
    add a, d
    ld a, (a)           // a = param1
    and a, a
    jzi baseCase

    // Recursive case: prepare arguments
    dec a               // a = param1 - 1
    dec d
    st (d), a           // push argument

    // Push return address
    ldi a, . 5 +
    dec d
    st (d), a

    // Recursive call
    jmpi func

    // Clean up argument
    inc d

    // Use return value in a
    // ... process result ...
    jmpi func_return

baseCase:
    ldi a, 1            // base case return value

func_return:
    // Deallocate locals
    ldi b, func_localVarSize
    add d, b

    // Return
    ld b, (d)
    inc d
    jmp b
```

1.8.8 Pattern 8: Passing Address of Variable

```
// To pass &x as an argument:
ldi a, x          // a = &x (don't dereference!)
dec d
st (d), a         // push &x
```

1.8.9 Pattern 9: Dereferencing a Pointer Parameter

```
// Function: void func(uint8_t *ptr)
// To access *ptr:

func:
    func_ptr: 1       // offset to ptr parameter

    ldi a, func_ptr
    add a, d
    ld a, (a)         // a = ptr (the address)
    ld b, (a)         // b = *ptr (the value at that address)
```

1.8.10 Pattern 10: String Literal

```
// String "Hi" with null terminator:
__lit_str1:
    byte 72           // 'H'
    byte 105          // 'i'
    byte 0            // null terminator

// To pass string to function:
ldi a, __lit_str1   // a = address of string
dec d
st (d), a           // push string address
```

1.9 Debugging Strategies

1.9.1 1. Trace the Stack

Most bugs are stack-related! Always track:

- Where does D point?
- What's at D+0, D+1, D+2, etc.?
- Is the stack balanced after function calls?

```
Before call:
D → [some_value]

After push arg2 (5):
D → [5]
    [some_value]

After push arg1 (3):
D → [3]
    [5]
    [some_value]

After push retAddr:
D → [retAddr]
    [3]
    [5]
    [some_value]

At function entry:
D → [retAddr]  ← callee sees this
    [arg1=3]   ← D+1
    [arg2=5]   ← D+2

After allocating 2 locals:
D → [local_a]  ← D+0
    [local_b]  ← D+1
    [retAddr]  ← D+2
    [arg1=3]   ← D+3
    [arg2=5]   ← D+4

After deallocating locals:
D → [retAddr]

After popping retAddr:
D → [arg1=3]

After caller cleans args:
D → [some_value]  ← BALANCED!
```

1.9.2 2. Check Offset Calculations

Common mistake: Wrong offset to parameters/locals

Verification Checklist:

Did you skip the return address when computing parameter offsets?
Are local variable offsets counting from 0?
Is `localVarSize` correct?
After allocating locals, do parameter offsets increase by `localVarSize`?

Example:

```
func:
    func_a: 0              // local var a at offset 0
    func_b: func_a + 1     // local var b at offset 1
    func_lvs: func_b + 1   // total local size = 2
    func_x: func_lvs + 1   // param x at offset 3 (2 locals + 1 retAddr)
    func_y: func_x + 1     // param y at offset 4
```

Verify: After allocating locals, D points to $\texttt{func}_a, so:$

$$\texttt{func}_a isatD + 0\texttt{func}_b isatD + 1 Returnaddressisat D + 2$$

$\texttt{func}_x isatD + 3\texttt{func}_y isatD + 4$

1.9.3 3. Verify Comparison Logic

Common mistake: Using wrong comparison operator

Debugging Checklist:

Is the comparison native (`<` or `==`) or derived?
For `>`, did you reverse operands to `<`?
For `!=`, did you negate `==` correctly?
For `<=` and `>=`, did you use OR of `<` and `==`?
Are you jumping on the right flag (jci for `<`, jzi for `==`)?

Example Bug:

```
// WRONG: Testing x > y
ldi a, x
ld a, (a)
ldi b, y
```

```
cmp a, b        //    This computes x - y
jci greater     //    This jumps if x < y, NOT x > y!
```

Fix:

```
// CORRECT: Testing x > y (reverse to y < x)
ldi a, y        //    Load y first
ld a, (a)
ldi b, x        //    Load x second
ld b, (b)
cmp a, b        //    Compute y - x
jci greater     //    Jump if y < x, i.e., x > y
```

1.9.4 4. Track Register Values

Registers A, B, C are **not preserved** across calls!

Example Bug:

```
ldi a, x
ld a, (a)       // a = x
// ... prepare to call function ...
jmpi someFunc
//      BUG: a is now overwritten by someFunc's return value!
// Can't assume a still contains x
```

Fix: Save to stack or use after call

```
ldi a, x
ld a, (a)       // a = x
dec d
st (d), a       //    Save x on stack
// ... call function ...
jmpi someFunc
ld a, (d)       //    Restore x from stack
inc d
```

1.9.5 5. Verify Return Address Calculation

Common mistake: Wrong offset in `. N +`

How to count:

```
ldi a, . 5 +    // 1     this instruction
dec d           // 2
st (d), a       // 3
jmpi func       // 4
inc d           // 5     return point (PC + 5)
inc d           // 6
// more code...
```

The number should be the count from `ldi a, . N +` to the **first instruction after the jump**.

1.9.6 6. Check Stack Balance

Rule: After a function call completes, D should point to the same location as before the call.

Verification:

```
// Before call: D = 0x0100
ldi a, 5
dec d           // D = 0x00FF
st (d), a
ldi a, 3
dec d           // D = 0x00FE
st (d), a
ldi a, . 5 +
dec d           // D = 0x00FD
st (d), a
jmpi func
// After func returns: D = 0x00FE (callee popped retAddr)
inc d           // D = 0x00FF
inc d           // D = 0x0100     BALANCED!
```

1.9.7 7. Trace Control Flow

For complex conditionals, trace each path:

```
// if ((x > 2) && (x != y)) goto label;

// Path 1: x <= 2
ldi a, 2
ldi b, x
ld b, (b)
cmp a, b        // 2 - x
jci skip        // if 2 < x (x > 2), skip to next check
jzi skip        // if 2 == x (x == 2), skip to next check
jmpi label      // x < 2, so condition is false, don't goto label
skip:

// Path 2: x > 2, now check x != y
ldi a, x
ld a, (a)
ldi b, y
ld b, (b)
cmp a, b        // x - y
jzi end         // if x == y, condition is false, don't goto label
jmpi label      // x != y and x > 2, condition is true, goto label
end:
```

Trace each scenario:

- x=1, y=5: Path 1, no jump
- x=2, y=5: Path 1, no jump
- x=3, y=3: Path 2, x==y, no jump
- x=3, y=5: Path 2, x!=y, jump

Common in:

- Loop conditions
- Array indexing
- Stack offset calculations

Example:

```
// Loop from i=0 to i<N (not i<=N!)
ldi a, 0          // i = 0
loopBegin:
ldi b, N
cmp a, b          // i - N
jzi loopEnd       //      BUG: jumps when i==N, but should also jump when i>N!
jci loopBody      //      Jump if i < N
jmpi loopEnd      //      Otherwise, end loop
loopBody:
// ... loop body ...
inc a             // i++
jmpi loopBegin
loopEnd:
```

Better:

```
loopBegin:
ldi b, N
cmp a, b          // i - N
jci loopBody      // if i < N, continue
jmpi loopEnd      // otherwise, exit
loopBody:
// ... loop body ...
inc a
jmpi loopBegin
loopEnd:
```

1.10 Common Mistakes \ How to Spot Them

1.10.1 Mistake 1: Forgetting to Dereference

Bug:

```
ldi a, x          // a = &x
add a, b          //      Adding to address, not value!
```

Fix:

```
ldi a, x          // a = &x
ld a, (a)         //      a = x (dereference!)
add a, b          //      Adding values
```

How to spot: Look for `ldi` loading a label, then using it directly in arithmetic.

1.10.2 Mistake 2: Wrong Argument Order

Bug:

```
// Calling func(3, 5) but pushing in wrong order
ldi a, 3
dec d
st (d), a         //      Pushed first argument first!
ldi a, 5
dec d
st (d), a
```

Fix:

```
// Push in REVERSE order (last argument first)
ldi a, 5          //      Second argument
dec d
st (d), a
ldi a, 3          //      First argument
dec d
st (d), a
```

How to spot: First argument should have **lower address** (pushed last).

1.10.3 Mistake 3: Callee Cleaning Up Arguments

Bug:

```
func:
    // ... function body ...
    ld b, (d)
    inc d             // pop return address
    inc d             //      Popping argument (caller's job!)
    inc d             //      Popping argument (caller's job!)
    jmp b
```

Fix:

```
func:
    // ... function body ...
    ld b, (d)
    inc d             //      Only pop return address
    jmp b             //      Caller will clean up arguments
```

How to spot: Callee should only `inc d` once (for return address).

1.10.4 Mistake 4: Not Preserving D

Bug:

```
func:
    ldi a, func_x
    add d, a          //      Modifying D directly!
    ld a, (d)
```

```
func:
    cpr c, d        //     Copy D to C
    ldi a, func_x
    add c, a        //     Modify C, not D
    ld a, (c)       //     Use C for access
```

How to spot: D should only change via `inc d`, `dec d`, `add d, localVarSize`, or `sub d, localVarSize`.

1.10.5 Mistake 5: Incorrect Return Address Offset

Bug:

```
ldi a, . 3 +    //     Wrong count
dec d
st (d), a
jmpi func
inc d           // Return point (actually 5 instructions away!)
inc d
```

Fix:

```
ldi a, . 5 +    //     Count: 1=ldi, 2=dec, 3=st, 4=jmpi, 5=inc
dec d
st (d), a
jmpi func
inc d           //     This is PC+5
inc d
```

How to spot: Count instructions from `ldi` to first instruction after `jmpi`.

1.10.6 Mistake 6: Forgetting to Set Return Value

Bug:

```
func:
    // ... computation ...
    ldi b, result
    ld b, (b)       //     Result in B, not A!
    ld c, (d)
    inc d
    jmp c
```

Fix:

```
func:
    // ... computation ...
    ldi a, result   //     Put result in A
    ld a, (a)
    ld b, (d)
    inc d
    jmp b
```

How to spot: Return value must be in register A.

1.10.7 Mistake 7: Unbalanced Stack

Bug:

```
// Caller
ldi a, 5
dec d
st (d), a       // Push 1 argument
ldi a, . 5 +
dec d
st (d), a       // Push return address
jmpi func
inc d           //     Only cleaned up 1 byte, but pushed 2!
```

Fix:

```
// Caller
ldi a, 5
dec d
st (d), a       // Push 1 argument
ldi a, . 5 +
dec d
st (d), a       // Push return address
jmpi func
inc d           //     Clean up argument
//     Stack is balanced (callee cleaned up retAddr)
```

How to spot: Count `dec d` before call, count `inc d` after call. Should differ by 1 (the return address).

1.10.8 Mistake 8: Using Wrong Jump Instruction

Bug:

```
cmp a, b
jci label       //     Jumps if a < b
// Intended: jump if a == b
```

Fix:

```
cmp a, b
jzi label       //     Jumps if a == b
```

How to spot:

- `jci` = jump if **carry** (less than)
- `jzi` = jump if **zero** (equal)

1.10.9 Mistake 9: Overwriting Return Address

Bug:

```
func:
```

```
    st (d), a        //      D points to return address!
```

Fix:

```
func:
    ldi a, func_lvs
    sub d, a         //      Allocate locals first
    ldi a, 10
    st (d), a        //      Now D points to local var
```

How to spot: Never `st (d), ...` at function entry before allocating locals.

1.10.10 Mistake 10: Infinite Loop

Bug:

```
loopBegin:
    // ... loop body ...
    jmpi loopBegin   //      No exit condition!
```

Fix:

```
loopBegin:
    // Check exit condition
    ldi a, counter
    ld a, (a)
    ldi b, limit
    cmp a, b
    jzi loopEnd      //      Exit when counter == limit

    // ... loop body ...
    jmpi loopBegin
loopEnd:
```

How to spot: Every loop must have a conditional jump out.

1.11 Practice Problems

1.11.1 Problem 1: Simple Function Call

Task: Implement `uint8`t *add(uint8t* `a, uint8`$_t b) returna + b$;

<details> <summary>Solution</summary>

```
add:
    add_a: 1         // offset to param a
    add_b: 2         // offset to param b

    cpr c, d         // save stack pointer
    ldi a, add_a
    add c, a
    ld a, (c)        // a = param a
    inc c
    ld b, (c)        // b = param b
    add a, b         // a = a + b

    ld b, (d)        // return
    inc d
    jmp b

// Calling: add(3, 7)
main:
    ldi a, 7
    dec d
    st (d), a        // push 7
    ldi a, 3
    dec d
    st (d), a        // push 3
    ldi a, . 5 +
    dec d
    st (d), a        // push retAddr
    jmpi add
    inc d            // clean up arg1
    inc d            // clean up arg2
    // a now contains 10
    halt
```

</details>

1.11.2 Problem 2: If-Else Statement

Task: Implement `if (x > 5) y = 1; else y = 2;`

<details> <summary>Solution</summary>

```
// if (x > 5) { y = 1; } else { y = 2; }
// Transform x > 5 to 5 < x

ldi a, 5
ldi b, x
ld b, (b)        // b = x
cmp a, b         // 5 - x
jci then         // if 5 < x (x > 5), goto then
// else block
ldi a, 2
ldi b, y
st (b), a        // y = 2
jmpi endIf
then:
ldi a, 1
ldi b, y
st (b), a        // y = 1
endIf:
```

1.11.3 Problem 3: While Loop

Task: Implement while (i < 10) sum += i; i++;

<details> <summary>Solution</summary>

```
loopBegin:
    ldi a, i
    ld a, (a)       // a = i
    ldi b, 10
    cmp a, b        // i - 10
    jci loopBody    // if i < 10, continue
    jmpi loopEnd    // else, exit
loopBody:
    // sum += i
    ldi a, sum
    ld a, (a)       // a = sum
    ldi b, i
    ld b, (b)       // b = i
    add a, b        // a = sum + i
    ldi b, sum
    st (b), a       // sum = sum + i

    // i++
    ldi a, i
    ld a, (a)       // a = i
    inc a           // a = i + 1
    ldi b, i
    st (b), a       // i = i + 1

    jmpi loopBegin
loopEnd:
```

</details>

1.11.4 Problem 4: Array Access

Task: Implement sum = arr[0] + arr[1] + arr[2] where arr is a byte array

<details> <summary>Solution</summary>

```
// sum = arr[0] + arr[1] + arr[2]

ldi a, arr      // a = &arr[0]
ld b, (a)       // b = arr[0]

inc a           // a = &arr[1]
ld c, (a)       // c = arr[1]
add b, c        // b = arr[0] + arr[1]

inc a           // a = &arr[2]
ld c, (a)       // c = arr[2]
add b, c        // b = arr[0] + arr[1] + arr[2]

ldi a, sum
st (a), b       // sum = result
```

</details>

1.11.5 Problem 5: Structure Access

Task: Given struct Point uint8t x; uint8t y; , implement p.y = p.x + 1

<details> <summary>Solution</summary>

```
Point_x: 0
Point_y: 1
Point_size: 2

// Assume: c = address of struct Point p

// Load p.x
ldi a, Point_x
add a, c        // a = &p.x
ld b, (a)       // b = p.x

// Compute p.x + 1
inc b           // b = p.x + 1

// Store to p.y
ldi a, Point_y
add a, c        // a = &p.y
st (a), b       // p.y = p.x + 1
```

</details>

1.11.6 Problem 6: Recursive Factorial

Task: Implement uint8t fact(uint8t n) return (n == 0) ? 1 : n * fact(n-1);

Note: Since TTPASM has no multiply, use repeated addition or simplify to n + fact(n-1) for practice.

<details> <summary>Solution (simplified: n + fact(n-1))</summary>

```
fact:
    fact_n: 1

    // Load n
    ldi a, fact_n
    add a, d
    ld a, (a)       // a = n

    // Check if n == 0
    and a, a
    jzi baseCase
```

```
    // Recursive case: n + fact(n-1)
    // Save n
    dec d
    st (d), a       // push n

    // Prepare fact(n-1)
    dec a           // a = n - 1
    dec d
    st (d), a       // push n-1

    ldi a, . 5 +
    dec d
    st (d), a       // push retAddr

    jmpi fact

    // Clean up argument
    inc d

    // a = fact(n-1), now add n
    ld b, (d)       // b = n (saved earlier)
    inc d           // pop saved n
    add a, b        // a = n + fact(n-1)

    jmpi fact_return

baseCase:
    ldi a, 1        // return 1

fact_return:
    ld b, (d)
    inc d
    jmp b
```

</details>

1.11.7 Problem 7: Pointer Dereferencing

Task: Implement void swap(uint8t *px, uint8t* py) that swaps values

<details> <summary>Solution</summary>

```
swap:
    swap_t: 0       // local var t
    swap_lvs: 1     // 1 byte for local
    swap_px: 2      // param px
    swap_py: 3      // param py

    // Allocate local
    ldi a, swap_lvs
    sub d, a

    // t = *px
    ldi a, swap_px
    add a, d
    ld a, (a)       // a = px (address)
    ld b, (a)       // b = *px (value)
    ldi a, swap_t
    add a, d
    st (a), b       // t = *px

    // *px = *py
    ldi a, swap_py
    add a, d
    ld a, (a)       // a = py (address)
    ld b, (a)       // b = *py (value)
    ldi a, swap_px
    add a, d
    ld a, (a)       // a = px (address)
    st (a), b       // *px = *py

    // *py = t
    ldi a, swap_t
    add a, d
    ld b, (a)       // b = t
    ldi a, swap_py
    add a, d
    ld a, (a)       // a = py (address)
    st (a), b       // *py = t

    // Deallocate and return
    ldi a, swap_lvs
    add d, a
    ld b, (d)
    inc d
    jmp b
```

</details>

1.11.8 Problem 8: Debugging Challenge

Task: Find the bug in this code that should compute x = 2 * y

```
// Bug version
ldi a, y
ld a, (a)       // a = y
add a, a        // a = 2 * y
ldi b, x
ld b, (b)       //     BUG HERE!
st (b), a       // x = 2 * y
```

Bug: Line 5 loads the value of x, but we want the address!

Fix:

```
1 ldi a, y
2 ld a, (a)       // a = y
3 add a, a        // a = 2 * y
4 ldi b, x        //     b = &x (don't dereference!)
5 st (b), a       //     x = 2 * y
```

</details>

1.12 Quick Reference Card

1.12.1 Stack Operations

```
// Push register X
dec d
st (d), x

// Pop to register X
ld x, (d)
inc d
```

1.12.2 Comparison Operations

```
// x < y
cmp x, y
jci label

// x == y
cmp x, y
jzi label

// x > y (reverse to y < x)
cmp y, x
jci label

// x != y
cmp x, y
jzi skip
jmpi label
skip:
```

1.12.3 Function Call Template

```
// Caller
ldi a, arg2
dec d
st (d), a
ldi a, arg1
dec d
st (d), a
ldi a, . 5 +
dec d
st (d), a
jmpi func
inc d           // clean arg1
inc d           // clean arg2
```

1.12.4 Function Definition Template

```
func:
    func_local1: 0
    func_lvs: 1
    func_param1: 2
    func_param2: 3

    ldi a, func_lvs
    sub d, a        // allocate locals

    // ... function body ...

    ldi a, func_lvs
    add d, a        // deallocate locals
    ld b, (d)
    inc d
    jmp b
```

1.12.5 Memory Access

```
// Load global var
ldi a, var
ld a, (a)

// Store to global var
ldi a, var
st (a), value_reg

// Access frame item
ldi a, offset
add a, d
ld/st (a), ...
```

1.13 Final Exam Tips

1. Draw the stack for every function call
2. Count instructions carefully for return addresses
3. Verify offsets match the stack layout
4. Check stack balance before and after calls

6. Remember: Arguments in reverse, caller cleans up
7. Remember: Return value in A, A/B/C not preserved
8. Remember: Only < and == are native comparisons
9. Test edge cases: zero, negative (if signed), boundary values
10. Read the C comments - they tell you the intent!

1.14 Good Luck!

Remember: TTPASM is simple but requires careful attention to detail. Most bugs come from:

- Stack mismanagement
- Wrong offsets
- Incorrect comparison logic
- Unbalanced stack

Practice tracing code by hand - this is the best way to prepare!