TANG NANO 9K / GW1NR-9

CPU8

Systemdokumentation

ISA, Hardware, Speicher, UART-Debugger und Toolchain

Dokumentationsstand	15. November 2026
RTL-Basis	Git-Stand 12d7238 plus aktueller LED-MMIO-Arbeitsstand
Zielplattform	Sipeed Tang Nano 9K
Systemprofil	8 Bit / drei Phasen / LED-MMIO bei 0xFF

Technische Referenz aus RTL, Tests, C-Tools und Python-Debugger

Inhalt

1 Zweck und Systemstatus ... 3
1.1 Kerndaten ... 3
1.2 Bewusste Architekturentscheidungen ... 3
2 Gesamtarchitektur ... 4
2.1 Blockschaltbild des FPGA-Systems ... 4
2.2 CPU-interne Datenwege ... 4
2.2.1 Register und Zustände ... 5
3 Ausführung und Timing ... 6
3.1 Drei Phasen pro Instruktion ... 6
3.2 Timing einer normalen Instruktion ... 6
3.3 Sonderfall GET ... 6
3.4 Sonderfall JMP ... 6
3.5 Halt und Clock-Enable ... 7
4 Instruction Set Architecture ... 8
4.1 Kodierung ... 8
4.2 Adressierungsmodell ... 8
4.3 Resetwerte ... 9
5 Speicher und Memory-Mapped I/O ... 10
5.1 Getrennte Adressräume ... 10
5.2 MMIO-Datenfluss ... 10
5.2.1 Schreiben ... 10
5.2.2 Lesen ... 10
5.2.3 Board-Ausgang ... 10
5.3 Beispiel: ALU-Ergebnis anzeigen ... 11
6 Hardwaremodule und FPGA-Integration ... 12
6.1 RTL-Hierarchie ... 12
6.2 Zentrale Module ... 12
6.3 Pins und Takt ... 12
7 UART-Debugger ... 13
7.1 Architektur und Zweck ... 13
7.2 Kommandopaket ... 13
7.3 Antwortpaket ... 13
7.4 Terminaloberfläche ... 14
7.4.1 Start ... 15
7.5 Debugger-Grenzen ... 15
8 Programmassembler ... 16
8.1 Aufgabe und Build ... 16
8.2 Unterstützte Syntax ... 16
8.3 Byteerzeugung ... 16
8.4 Aktuelle Einschränkungen ... 17
9 Memory-/Heap-Assembler ... 18
9.1 Aufgabe ... 18
9.2 Syntax ... 18
9.3 Ausgabe und Überschreiben ... 18
10 Build, Test und FPGA-Workflow ... 19
10.1 Typischer Ablauf ... 19
10.2 Initialisierungsdateien im FPGA-Build ... 19
10.3 Testabdeckung ... 19
11 Vollständiges Programmbeispiel ... 20
12 Bekannte Grenzen und nächste sinnvolle Schritte ... 21
13 Referenz: wichtige Dateien ... 22

Zweck und Systemstatus

Dieses Dokument beschreibt den implementierten Zustand des CPU8-Systems. Es ist eine Bestandsaufnahme des vorhandenen RTL und der dazugehörigen Werkzeuge, keine Beschreibung einer nur geplanten Architektur. Besonders berücksichtigt ist das neu hinzugefügte, speicherabgebildete LED-Register.

> **Kurzfassung**
> CPU8 ist eine 8-Bit-Akkumulator-ähnliche Eigenbau-CPU mit getrenntem Programm-ROM und Daten-RAM. Adressen und Daten sind jeweils acht Bit breit. Instruktionen laufen in den drei Phasen FETCH, COMBINE und COMMIT. Das MAR wird explizit durch `GET` geladen und anschließend von `LDA`, `LDB`, `STR` und dem indirekten `JMP` verwendet.

Kerndaten

Eigenschaft	Aktueller Stand
Datenbreite	8 Bit
Adressbreite	8 Bit für Programm- und Datenraum
Programmraum	256 x 8 Bit, synchron gelesen, eigener Controller
Datenraum	256 x 8 Bit physischer RAM; CPU-seitig `0x00` bis `0xFE` als RAM verfügbar
MMIO	`0xFF`: sechs Bit breites LED-Register
Register	PC, IR, MAR, A, B, RESULT sowie Phasenzähler und Priming-Status
Instruktionssatz	17 Opcodes (`0x00` bis `0x10`)
Instruktionslänge	Ein Byte, außer `GET`: Opcode plus ein Adressbyte
Takt	27 MHz Systemtakt; CPU-Fortschritt ausschließlich über Clock-Enable
Debug-Schnittstelle	UART 115200 Baud, 8N1, binäres Request/Response-Protokoll
Board-Ausgänge	Sechs aktive-low LEDs, softwaregesteuert über MMIO

Bewusste Architekturentscheidungen

- Harvard-Aufteilung: Programm-ROM und Daten-RAM besitzen unabhängige Controller und Busse.
- Kein abgeleiteter CPU-Takt: Debugger und Run-Modus erzeugen ein `cpu_enable`, nicht einen gegateten Takt.
- Sticky MAR: Eine mit `GET` gesetzte Adresse bleibt aktiv, bis ein weiteres `GET` ausgeführt wird; auch `JMP` verwendet diese Adresse.
- Kein unmittelbarer Operand für `LDA`, `LDB` oder `STR`: Adressierung erfolgt immer indirekt über das MAR.
- LED-MMIO nutzt den normalen CPU-Datenpfad; nur innerhalb des Data-Memory-Controllers wird bei Adresse `0xFF` umgeschaltet.

Gesamtarchitektur

Blockschaltbild des FPGA-Systems

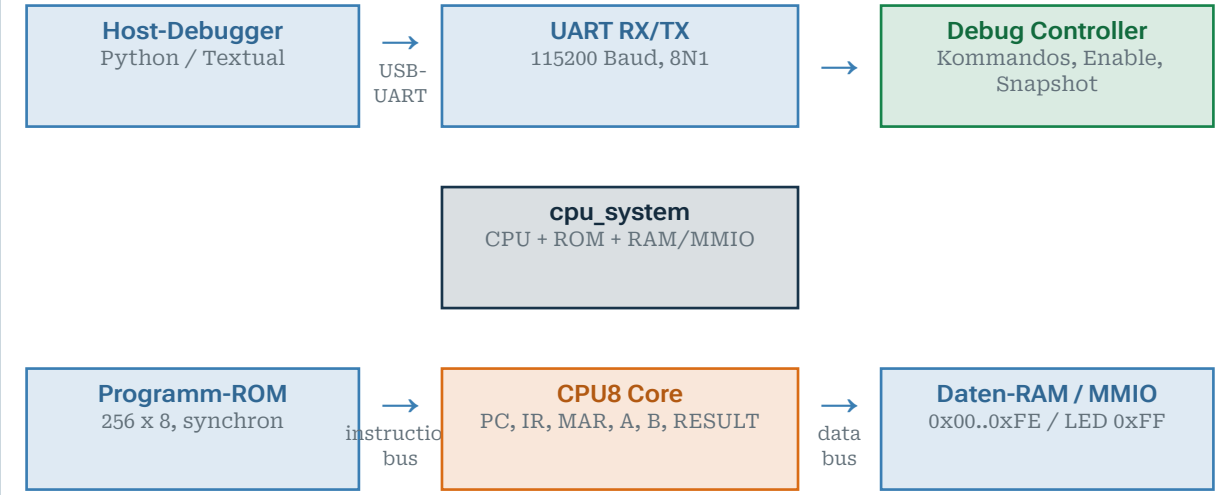


Abbildung 1: Systemübersicht vom Host bis zu CPU, Speichern und LED-Ausgang.

Die oberste RTL-Instanz `tang_nano_9k_top` verbindet den 27-MHz-Boardtakt, den aktiven-low Reset-Taster, UART und LEDs. Der Reset-Eingang wird über zwei Flipflops in die Taktdomäne synchronisiert. UART-Receiver, UART-Transmitter, Debug-Controller und CPU-System arbeiten alle im selben Systemtakt.

CPU-interne Datenwege

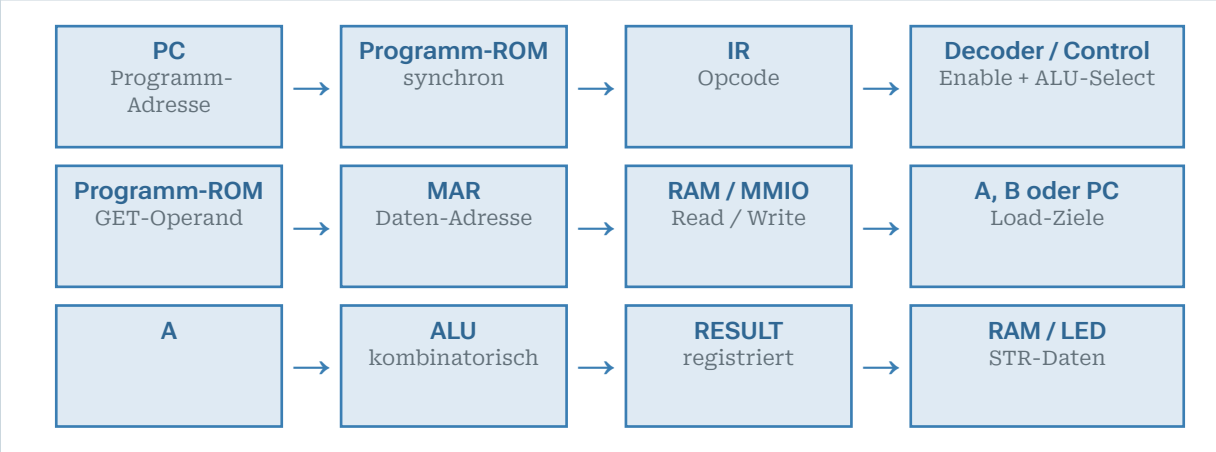


Abbildung 2: Wesentliche CPU-Datenpfade. Alle sichtbaren Zustandsregister werden an steigenden Taktflanken aktualisiert.

Register und Zustände

Register	Breite	Quelle	Ladebedingung
PC	8	eigener Zähler oder Datenbus	FETCH jeder Instruktion; zusätzlich COMBINE bei `GET`; COMMIT von `JMP`
IR	8	Programm-ROM-Ausgang	FETCH
MAR	8	Programm-ROM-Ausgang	COMMIT von `GET`
A	8	Datenbus-Eingang	COMMIT von `LDA`
B	8	Datenbus-Eingang	COMMIT von `LDB`
RESULT	8	ALU-Ausgang	COMMIT einer ALU-Instruktion
cycle	2	Phasenzähler	`00 → 01 → 10 → 00` bei `cpu_enable`
fetch_valid	1	Priming-Status	erstes Enable nach Reset
LED	6	`RESULT[5:0]`	MMIO-Schreibzugriff auf `0xFF`

Der ALU-Fehlerausgang berechnet Unsigned-Überlauf/Unterlauf und Signed-Overflow für arithmetische Operationen. Im aktuellen CPU-System wird dieses Signal jedoch weder in einem Statusregister gespeichert noch über den Debugger ausgegeben.

Ausführung und Timing

Drei Phasen pro Instruktion

Phase	Code	Funktion
FETCH	`00`	IR übernimmt den vorab gelesenen ROM-Wert. Der PC wird inkrementiert.
COMBINE	`01`	ALU arbeitet kombinatorisch. `LDA/LDB/JMP` lösen den synchronen Daten-Read aus. `GET` liest das Operandenbyte und inkrementiert den PC ein zweites Mal.
COMMIT	`10`	Zielregister oder Speicher wird geschrieben: A, B, RESULT, MAR, PC oder RAM/MMIO.

Nach Reset steht die CPU in FETCH, der synchrone ROM-Ausgang ist aber noch nicht gültig. Ein gesonderter Priming-Schritt liest deshalb ROM-Adresse `0x00`, ohne PC oder Phase weiterzuschalten. Erst danach ist `fetch_valid = 1` und ein normaler Phasen- oder Instruktionsschritt möglich.

Timing einer normalen Instruktion

```
Flanke          Priming       FETCH          COMBINE        COMMIT
Phase           00            00 -> 01       01 -> 10       10 -> 00
ROM             ROM[PC]       nächstes Byte  nächstes Byte  nächstes Byte
IR              hält          <- ROM         hält           hält
PC              hält          PC + 1         hält           hält
Zielregister    hält          hält           vorbereitet    wird geschrieben
```

Sonderfall GET

`GET` ist die einzige Instruktion mit einem zusätzlichen Programmbyte:

```
Adresse im ROM:   N          N+1          N+2
Inhalt:           0F         Datenadresse nächster Opcode

FETCH:    IR <- 0F, PC: N -> N+1
COMBINE:  ROM liest Operand, PC: N+1 -> N+2
COMMIT:   MAR <- Operand
```

Durch Nonblocking Assignments sieht das synchrone ROM an einer Flanke jeweils den alten PC-Wert. Das zusätzliche PC-Inkrement in COMBINE stellt sicher, dass der Operand nicht als Opcode ausgeführt wird.

Sonderfall JMP

`JMP` ist ein indirekter Ein-Byte-Sprung über die aktuelle MAR-Adresse. Er verhält sich auf dem Datenpfad wie eine Load-Instruktion mit dem PC als Ziel:

```
Vorbereitung:  GET aa          MAR <- aa
COMBINE:       Daten-Read      Ziel <- DATA[MAR]
COMMIT:        PC <- Ziel      fetch_valid <- 0
Priming:       ROM liest PC    PC und Phase halten
FETCH:         IR <- ROM[PC]   normale Ausführung läuft weiter
```

Der zusätzliche Priming-Takt ist notwendig, weil das synchrone Programm-ROM beim PC-Load noch das sequentiell vorab gelesene Byte ausgibt. Durch `fetch_valid = 0` wird dieses Byte verworfen und zuerst die neue Zieladresse gelesen.

Halt und Clock-Enable

Bei `cpu_enable = 0` halten CPU-Register und Phasenzähler ihren Zustand. Gleichzeitig werden Programm-Read, Daten-Read und Daten-Write unterdrückt. Eine in COMMIT angehaltene `STR`-Instruktion schreibt daher noch nicht; erst ein späteres Enable erzeugt genau den vorgesehenen Schreibimpuls.

Instruction Set Architecture

Kodierung

Der Opcode belegt immer ein vollständiges Byte. Die 17 definierten Werte liegen im Bereich `0x00` bis `0x10`. Nur `GET` besitzt ein zweites Byte.

Hex	Binär	Mnemonic	Bytes	Semantik
00	0000 0000	`NOP`	1	Keine Zustandsänderung außer normalem PC-/Phasenfortschritt.
01	0000 0001	`LDA`	1	`A <- DATA[MAR]`
02	0000 0010	`LDB`	1	`B <- DATA[MAR]`
03	0000 0011	`STR`	1	`DATA[MAR] <- RESULT`
04	0000 0100	`ADD`	1	`RESULT <- A + B` modulo 256
05	0000 0101	`AND`	1	`RESULT <- A AND B`
06	0000 0110	`NOT`	1	`RESULT <- NOT A`
07	0000 0111	`OR`	1	`RESULT <- A OR B`
08	0000 1000	`SUB`	1	`RESULT <- A - B` modulo 256
09	0000 1001	`XOR`	1	`RESULT <- A XOR B`
0A	0000 1010	`SHL`	1	`RESULT <- A << 1`, Bit 0 wird null
0B	0000 1011	`ROL`	1	`RESULT <- rotate_left(A, 1)`
0C	0000 1100	`SHR`	1	Logischer Shift rechts; Bit 7 wird null
0D	0000 1101	`ROR`	1	Rotation rechts um ein Bit
0E	0000 1110	`TWC`	1	`RESULT <- 0 - A` (Zweierkomplement)
0F	0000 1111	`GET aa`	2	`MAR <- aa`; `aa` ist das folgende ROM-Byte
10	0001 0000	`JMP`	1	`PC <- DATA[MAR]`; anschließend ROM-Priming

Adressierungsmodell

`LDA`, `LDB`, `STR` und `JMP` enthalten keine Adresse. Sie greifen auf die zuletzt in MAR geladene Adresse zu:

```
GET 0x20    # MAR := 0x20
LDA         # A := DATA[0x20]

GET 0x21    # MAR := 0x21
LDB         # B := DATA[0x21]

ADD         # RESULT := A + B
GET 0x22    # MAR := 0x22
STR         # DATA[0x22] := RESULT

GET 0x30    # MAR := 0x30
JMP         # PC := DATA[0x30]
```

Wichtig: MAR bleibt erhalten
Zwischen `GET` und einer Speicher- oder Sprunginstruktion dürfen beliebige ALU-Instruktionen liegen. Solange kein weiteres `GET` ausgeführt wird, bleibt dieselbe Datenadresse aktiv. Ein nacktes `STR` schreibt und ein nacktes `JMP` springt daher immer über die zuletzt gesetzte MAR-Adresse.

Resetwerte

PC, IR, MAR, A, B, RESULT, Phase und LED-Register werden synchron auf null gesetzt. Der Inhalt von Daten-RAM und Programm-ROM wird durch die jeweiligen Hex-Images bestimmt; ohne Daten-Initialisierungsdatei ist der RAM-Inhalt vor dem ersten Schreiben nicht spezifiziert.

Speicher und Memory-Mapped I/O

Getrennte Adressräume

Bereich	Adressen	Breite	Initialisierung
Programm-ROM	`0x00..0xFF`	256 x 8	`program/out.hex`
Daten-RAM	`0x00..0xFE`	255 CPU-erreichbare Bytes	`heap/out.hex`
LED-MMIO	`0xFF`	6-Bit-Register	Reset auf `0x00`

Physisch ist der Daten-RAM weiterhin als Array mit 256 Bytes deklariert. Der Data-Memory-Controller dekodiert jedoch `0xFF` vor dem Zugriff. CPU-Zugriffe auf diese Adresse erreichen ausschließlich das MMIO-Register, nicht RAM-Zelle 255.

MMIO-Datenfluss



Abbildung 3: Der CPU-seitige Bus bleibt unverändert. Der Controller wählt anhand der Adresse zwischen RAM und MMIO.

Schreiben

Bei `STR` und `MAR = 0xFF` wird der RAM-Schreibpfad gesperrt. `mmio_write_enable` wird aktiv und `RESULT[5:0]` wird in das LED-Register übernommen. Die Bits 7 und 6 werden verworfen.

```
RESULT[7:0] = xx_abcdef
                    |
                    +--> LED-Register[5:0] = abcdef
```

Lesen

Bei `LDA`, `LDB` oder `JMP` und `MAR = 0xFF` liefert der Controller den LED-Wert synchron und nullerweitert auf dem normalen 8-Bit-Datenbus. Für `JMP` wird dieser Wert als neue Programmadresse verwendet:

```
LED[5:0] = abcdef  -->  CPU-Datenbus = 00_abcdef
```

Board-Ausgang

Die sechs LEDs des Tang Nano 9K sind aktive-low. Das Top-Level invertiert deshalb das logische Register:

```
assign led = ~led_data;
```

Ein Registerwert `0` schaltet alle LEDs aus; gesetzte Registerbits schalten die entsprechenden LEDs ein.

Beispiel: ALU-Ergebnis anzeigen

```
GET 0x0D
LDA
GET 0x0C
LDB
SUB
GET 0xFF
STR
```

Das Ergebnis von `A - B` landet zuerst in RESULT. `GET 0xFF` setzt anschließend MAR auf die MMIO-Adresse; `STR` übernimmt die unteren sechs Ergebnisbits in das LED-Register.

Hardwaremodule und FPGA-Integration

RTL-Hierarchie

```
tang_nano_9k_top
|- reset synchronizer
|- uart_receiver
|- uart_transmitter
|- cpu_debug_controller
`- cpu_system
   |- cpu_8
   |  |- opcode_decoder
   |  |- control_unit
   |  |- cycle_controller
   |  |- PC / fetch_valid
   |  |- IR / MAR / A / B / RESULT registers
   |  `- arithmetic_logic_unit_8
   |- program_memory_controller
   |- data_memory_controller
   `- LED register
```

Zentrale Module

Modul	Aufgabe
`tang_nano_9k_top`	Board-Pins, Reset-Synchronisierung, UART, Debug-Controller, CPU-System und LED-Invertierung.
`cpu_system`	Verbindet CPU, Programm-ROM, Daten-Controller und LED-MMIO-Register.
`cpu_8`	Register, Busse, Decoder, Steuerwerk, Phasenzähler und ALU.
`control_unit`	Erzeugt Lade-, PC-, Speicher- und ALU-Steuersignale aus Opcode und Phase.
`program_memory_controller`	Synchroner 256-x-8-ROM-Read mit `$readmemh`.
`data_memory_controller`	Synchroner RAM-Zugriff plus Dekodierung und Routing der MMIO-Adresse.
`cpu_debug_controller`	UART-Kommandos, Run/Halt, Schritte, Reset/Priming und Snapshot-Erzeugung.

Pins und Takt

Signal	Pin	Richtung	Bedeutung
`sys_clk`	52	Eingang	27 MHz, Timing-Constraint 37,037 ns
`sys_rst_n`	4	Eingang	Aktiver-low Taster, intern synchronisiert
`uart_tx`	17	Ausgang	FPGA zum BL702/Host
`uart_rx`	18	Eingang	BL702/Host zum FPGA
`led[0]`	10	Ausgang	Aktive-low LED
`led[1]`	11	Ausgang	Aktive-low LED
`led[2..5]`	13, 14, 15, 16	Ausgang	Aktive-low LEDs

Zielbaustein ist `GW1NR-LV9QN88PC6/I5`. Programm- und Datenspeicher tragen Attribute zur Gowin-BSRAM-Inferenz.

UART-Debugger

Architektur und Zweck

Der Debugger besteht aus drei Schichten:

1. UART-RX/TX im FPGA transportieren Bytes mit 115200 Baud, 8 Datenbits, keiner Parität und einem Stopbit.
2. `cpu_debug_controller` interpretiert Kommandos, erzeugt CPU-Enable-Impulse und friert Zustände in Antwortregistern ein.
3. Der Python-Client übernimmt Portauswahl, Framing, Prüfsumme und eine Textual-Terminaloberfläche.

Der Debugger verändert den Systemtakt nicht. Ein Phasenschritt entspricht genau einem Takt mit aktivem `cpu_enable`. Im Run-Modus erzeugt der Teiler drei Enable-Impulse pro Sekunde, also eine vollständige Instruktion pro Sekunde.

Nach `JMP` meldet der Instruktionsschritt die FETCH-Grenze zunächst als nicht geprimt, weil der PC-Redirect den sequentiellen ROM-Prefetch verwirft. Ein anschließender Phasenschritt primt nur das Sprungziel; ein Instruktionsschritt führt dieses Priming automatisch aus und läuft danach durch die Zielinstruktion bis COMMIT.

Kommandopaket

Jedes Hostkommando besteht aus zwei Bytes:

```
A5 CMD
```

Ein erneutes `A5` während eines unvollständigen Pakets startet die Synchronisierung neu.

CMD	Name	Wirkung
`01`	Phase step	Eine CPU-Phase ausführen, danach Zustand erfassen.
`02`	Instruction step	Bis einschließlich COMMIT laufen und an der nächsten FETCH-Grenze stoppen.
`03`	Run	Periodischen Run-Modus aktivieren.
`04`	Halt	Run-Modus stoppen; keine CPU-Phase ausführen.
`05`	Reset	CPU-Reset auslösen, ROM primen und in FETCH anhalten.
`06`	Read snapshot	Zustand lesen, ohne die CPU zu verändern.

Antwortpaket

Jedes akzeptierte oder abgelehnte Kommando erzeugt genau zwölf Bytes:

```
5A CMD STATUS FLAGS PC IR MAR A B RESULT PHASE CHECKSUM
```

Feld	Byte	Bedeutung
SYNC	`5A`	Startmarke der Antwort
CMD	variabel	Echo des bearbeiteten Kommandos
STATUS	`00/01/02`	OK / unbekanntes Kommando / UART-Framingfehler
FLAGS	Bitfeld	Bit 0 = RUN, Bit 1 = ROM primed
PC..RESULT	je 8 Bit	Atomar erfasster Registerzustand
PHASE	`00..02`	FETCH / COMBINE / COMMIT
CHECKSUM	XOR	XOR der vorhergehenden elf Bytes

Der Snapshot wird an einer Systemtaktgrenze in dedizierte Antwortregister kopiert. Während die zwölf UART-Bytes versendet werden, bleibt dieser Snapshot stabil, auch wenn die CPU im Run-Modus weiterläuft.

Terminaloberfläche

Der Python-Debugger zeigt PC, IR, MAR, A, B, RESULT, Phase, Run/Halt und Priming-Status. Ab 76 Spalten und 28 Zeilen wird zusätzlich eine Architekturansicht eingeblendet. Sie markiert Reads, kombinatorische Pfade und tatsächliche Register- oder Speicherwrites abhängig von Opcode und Phase.

Taste	Aktion
Leertaste	Eine Phase
`I`	Aktuelle Instruktion vervollständigen
`R`	Run/Halt umschalten
`X`	Reset, ROM-Priming, Halt
`Q`	Debugger beenden

Start

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install ./debugger
cpu8-debugger
```

Optional können Port, Baudrate und Timeout gesetzt werden:

```
cpu8-debugger --port /dev/cu.usbmodemXXXX --baud 115200 --timeout 0.5
```

Der Client erkennt einzelne passende `/dev/cu.usbmodem*`-, Sipeed- oder BL702-Ports automatisch. Bei keinem oder mehreren Kandidaten ist `--port` erforderlich.

Debugger-Grenzen

- Keine Breakpoints.
- Keine direkte RAM- oder ROM-Inspektion über das Protokoll.
- Keine Anzeige des LED-MMIO-Registers als eigenes Snapshot-Feld.
- Keine Anzeige oder Speicherung des ALU-Fehlerflags.
- Kommandos werden seriell abgearbeitet; der Host wartet auf die vollständige Antwort.

Programmassembler

Aufgabe und Build

Der C-basierte Programmassembler befindet sich unter `memory_tools/progmem`. Er liest eine textuelle Assemblerdatei und erzeugt im selben Verzeichnis eine `out.hex`. Jede ausgegebene Zeile enthält genau ein ROM-Byte in zweistelliger Hex-Schreibweise.

```
make build-progmem
make generate-progmem
```

Direkter Aufruf:

```
build/progmem program/test.asm
```

Unterstützte Syntax

Jede Eingabezeile entspricht genau einer Instruktion. Die Mnemonics werden anhand der ersten drei Zeichen erkannt und sind derzeit großgeschrieben zu verwenden.

```
NOP
LDA
LDB
STR
ADD
AND
NOT
OR
SUB
XOR
SHL
ROL
SHR
ROR
TWC
GET 0x2A
JMP
```

Für `GET` ist eine Hex-Adresse mit Präfix `0x` oder `0X` zwingend. Der erlaubte Bereich ist `0x00` bis `0xFF`; fehlende oder zusätzliche Zeichen werden für den GET-Operanden als Fehler gemeldet.

Byteerzeugung

Quelle	Ausgabe	Kommentar
`ADD`	`04`	Ein Byte
`STR`	`03`	Ein Byte; Adresse kommt aus MAR
`GET 0xFF`	`0F` / `FF`	Zwei Bytes in aufsteigenden ROM-Adressen
`JMP`	`10`	Ein Byte; Zieladresse kommt aus `DATA[MAR]`

Aktuelle Einschränkungen

> **Operanden bei STR werden nicht umgesetzt**
> `STR 0xFF` ist aktuell keine adressierte Store-Instruktion. Der Parser erkennt nur `STR` und gibt `03` aus; der Text hinter dem Mnemonic wird bei Nicht-GET-Instruktionen derzeit nicht validiert. Die CPU schreibt dadurch an die alte MAR-Adresse. Korrekt ist `GET 0xFF` gefolgt von `STR`.

Weitere Grenzen:

- Keine Labels oder symbolischen Konstanten.
- Keine Kommentare oder Leerzeilen als definierte Sprachkonstrukte.
- Keine Direktiven wie `.org`, `.byte` oder Includes.
- Keine automatische Auffüllung des ROM-Images auf 256 Zeilen.
- Keine adressierten Pseudoinstruktionen wie `LDA 0x20` oder `STR 0xFF`.
- Fehlende strikte Restzeichenprüfung bei allen Ein-Byte-Instruktionen.

Eine sinnvolle spätere Erweiterung wäre ein variabler Byte-Emitter. Dann könnte `STR 0xFF` als Pseudoinstruktion zu `0F FF 03` expandieren, ohne die Hardware-ISA zu verändern.

Memory-/Heap-Assembler

Aufgabe

`memory_tools/heap` erzeugt das Daten-RAM-Initialisierungsimage. Im Gegensatz zum Programmassembler arbeitet es adressorientiert: Jede Quellzeile weist einem Byte im 256-Byte-Image einen Wert zu. Nicht genannte Adressen bleiben null.

```
make build-heap
make generate-heap
```

Direkter Aufruf:

```
build/heapgen heap/heap.src
```

Syntax

```
# Kommentar
0C:05
0D:07
20 : FF
```

Adresse und Wert werden hexadezimal interpretiert und müssen zwischen `00` und `FF` liegen. Leerraum um Adresse, Doppelpunkt und Wert ist erlaubt. Leere Zeilen und Zeilen, deren erstes Nicht-Leerzeichen `#` ist, werden ignoriert. Zusätzliche Zeichen nach dem Wert führen zu einem Fehler.

Ausgabe und Überschreiben

Das Werkzeug initialisiert intern ein 256-Byte-Array mit null, verarbeitet die Einträge der Reihe nach und schreibt anschließend exakt 256 Hex-Zeilen nach `out.hex`. Wird dieselbe Adresse mehrfach angegeben, gewinnt der letzte Eintrag.

> **Adresse 0xFF im Heap-Image**
> Das Werkzeug kann zwar einen Initialwert für RAM-Zelle `0xFF` erzeugen, die CPU erreicht diese Zelle im aktuellen System nicht, weil Adresse `0xFF` als LED-MMIO dekodiert wird. Das LED-Register startet unabhängig vom Heap-Image mit null.

Build, Test und FPGA-Workflow

Typischer Ablauf

```
# C-Werkzeuge bauen
make tools

# program/out.hex und heap/out.hex erzeugen
make generate-images

# RTL- und Hosttests
make test

# Gowin-Bitstream bauen
make fpga
```

Der FPGA-Build startet `gw_sh` über `hardware/fpga/build.tcl`. Ausgabebasis ist `tang_nano_9k_cpu`; das resultierende Bitstream-File liegt typischerweise unter `hardware/impl/pnr/`.

Initialisierungsdateien im FPGA-Build

Da Gowin die Synthese aus einem Unterverzeichnis startet, verwendet das Top-Level relative Pfade:

```
Programm-ROM: ../../../program/out.hex
Daten-RAM:    ../../../heap/out.hex
```

Änderungen an `program/test.asm` oder `heap/heap.src` werden durch `make fpga` automatisch in neue Images übersetzt, weil das FPGA-Ziel von `generate-images` abhängt.

Testabdeckung

Ziel	Abdeckung
`test-alu`	Alle 16 ALU-Select-Werte über sämtliche 65.536 A/B-Kombinationen.
`test-adder-subtractor`	Addition und Subtraktion über sämtliche Eingangskombinationen.
`test-cpu`	Reset, Priming, GET, Loads, ALU und Enable-Verhalten.
`test-system`	ROM/RAM-Integration, GET-/JMP-Timing, Sprungziel-Priming, Halt/Resume, PC-Wraparound und LED-MMIO inklusive Readback.
`test-memory`	Synchrone Read-/Write-Flanken, Hold-Verhalten, MMIO-Routing und ungültiges gleichzeitiges Read/Write.
`test-uart`	8N1 Loopback von Transmitter und Receiver.
`test-debug`	Phasen-/Instruktionsschritt, Reset/Priming, Run/Halt, Snapshots und Fehlerantworten.
`test-host`	Python-Protokoll, Checksummen, Timeouts, Resynchronisierung und Portfilter.

Vollständiges Programmbeispiel

Das folgende Programm lädt zwei Werte, subtrahiert sie, schreibt das Ergebnis in den RAM und zeigt anschließend die unteren sechs Bits auf den LEDs:

```
# Annahme: heap.src enthält 0C:05 und 0D:07

GET 0x0D
LDA
GET 0x0C
LDB
SUB

GET 0x20
STR

GET 0xFF
STR
```

Erwarteter Ablauf:

1. `A = DATA[0x0D] = 0x07`.
2. `B = DATA[0x0C] = 0x05`.
3. `RESULT = 0x07 - 0x05 = 0x02`.
4. RAM-Adresse `0x20` erhält `0x02`.
5. LED-Register erhält `000010`; LED 1 leuchtet logisch.

Der Programmassembler unterstützt im aktuellen Zustand keine Kommentarzeilen. Für eine tatsächlich assemblierbare Datei müssen die Kommentar- und Leerzeilen aus diesem erklärenden Beispiel entfernt werden.

Bekannte Grenzen und nächste sinnvolle Schritte

Bereich	Aktuelle Grenze	Naheliegende Erweiterung
Assembler	Keine Labels, Direktiven oder Pseudoinstruktionen.	Tokenbasierter Parser und variabler Byte-Emitter.
Adressierung	Speicheroperationen benötigen vorheriges `GET`.	`LDA addr`, `LDB addr`, `STR addr` als reine Assembler-Makros.
Statusflags	ALU-Fehler wird nicht gespeichert.	Flags-Register plus Debug-Snapshot-Feld.
Debugger	Keine Breakpoints oder Speicheransicht.	Protokollkommandos für Read/Write und Breakpoints.
MMIO	Nur ein festes LED-Register bei `0xFF`.	Adressbereich und mehrere Peripherieregister definieren.
Programmfluss	Indirektes `JMP`, aber keine Bedingungen oder Stackoperationen.	Statusflags und bedingte Sprünge mit kontrollierter Opcode-Planung.
ROM-Tool	Image wird nicht auf 256 Bytes aufgefüllt.	Größenprüfung und deterministisches Padding.

Empfohlene Priorität
Der nächste risikoarme Schritt ist die Überarbeitung des Programmassemblers. Adressierte Pseudoinstruktionen verbessern die Lesbarkeit, ohne RTL oder ISA zu ändern. Gleichzeitig sollten zusätzliche Operanden bei Ein-Byte-Instruktionen nicht länger stillschweigend akzeptiert werden.

Referenz: wichtige Dateien

Pfad	Inhalt
`hardware/rtl/central_processing_unit.v`	CPU-Kern, Register und Busanschlüsse
`hardware/rtl/control_unit.v`	Phasenabhängige Steuersignale
`hardware/rtl/instruction_decoder.v`	Opcode-Dekodierung
`hardware/rtl/cpu_system.v`	Speicherintegration und LED-Register
`hardware/rtl/memory_controller.v`	Daten-RAM und MMIO-Routing
`hardware/rtl/tang_nano_9k_top.v`	Board-Top-Level
`hardware/rtl/cpu_debug_controller.v`	UART-Debug-Zustandsautomat
`memory_tools/progmem/`	Programmassembler
`memory_tools/heap/`	Datenimage-/Heap-Assembler
`debugger/cpu8_debugger/`	Python-Protokoll, Transport und Textual-UI
`hardware/tests/`	RTL-Testbenches
`Makefile`	Build-, Test-, Image- und FPGA-Ziele

Ende der Systemdokumentation