ChatGPT HarvOS Brainstorming Log

© 2025 Dennis Michael Heine. All rights reserved.

There is no "normal" Linux
for classic Harvard architecture CPUs.

Why?

- **Linux requires a von Neumann model**:
  The kernel expects code and data to be in the same address space, or at least to be flexibly accessible.
- **Harvard architecture strictly separates instruction and data memory**:
  Many classic Harvard CPUs (e.g. DSPs, older microcontrollers such as PIC, AVR) do not allow code to be treated like data, or only to a limited extent. This collides with central requirements of Linux (e.g. dynamic loading of programs, `mmap()`, `fork()`, shared libraries).

Exceptions & related:

- **Modified Harvard architectures** (e.g. ARM Cortex-M, AVR32):
  These are "Harvard-internal" (separate buses for data and instructions), but they have a standardized address model to the outside. Such CPUs can (in theory) run Linux - but in practice only if they have enough RAM/ROM. However, ARM Cortex-M is usually too small for this → **μClinux** (a slimmed-down variant for CPUs without MMU) is more likely to run there.
- **DSPs from TI (C6x series)**: There are **Linux ports** for some of them (as they have a kind of "Harvard+" with unified access).
- **Microcontroller (classic Harvard, e.g. PIC, AVR)**: Too small for Linux → **RTOS** (FreeRTOS, Zephyr, RIOT, etc.) is used instead.

Conclusion:

- **Pure Harvard architecture → not Linux**.
- **"Modified Harvard" (such as ARM, DSPs) → possible to a limited extent, partly with μClinux or customized ports.**

�Pure Harvard architectures (no Linux possible)

- **Atmel AVR (e.g. Arduino, ATmega, ATtiny)**
  - Strict separation of code and data memory.
  - No MMU, no unified address space.
  - → Only small RTOS such as **FreeRTOS, RIOT, Zephyr**.
- **Microchip PIC (PIC16, PIC18)**
  - Also real Harvard.
  - → Too small for Linux, only bare-metal or RTOS.
- **TI C2000 DSPs, older Harvard DSPs**
  - Very specialized, no Linux port known.

🟡 Modified Harvard architectures (partly Linux possible)

These architectures have separate buses, but a **uniform address space** or tricks so that Linux can run:

- **ARM Cortex-M (M0-M7)**
  - Internally Harvard, externally "Unified Memory".
  - No MMU → **Linux does not work**, but **μClinux** runs on some large M7s with enough RAM/flash.

    - Example: STM32F7/H7 can boot µClinux (very limited).
- **ARM Cortex-R**
    - Real-time ARM, Modified Harvard.
    - Also without MMU, but more power.
    - Some µClinux ports exist.
- **Analog Devices Blackfin DSP**
    - Modified Harvard, but unified addressing.
    - → There is an **official Linux port**!
    - Has been used in embedded devices (routers, audio hardware).

◍ Fully Harvard compatible with Linux

- **ARM Cortex-A (smartphones, SBCs)**
    - Internally often Harvard-like (separate I/D caches), but from a software perspective a completely normal von Neumann system.
    - → Normal Linux, Android, etc.
- **TI C6x DSPs (TMS320C6xxx)**
    - "Virtual unified memory access".
    - Linux ports exist, but exotic.
- **Xtensa (Espressif ESP32)**
    - Technically modified Harvard.
    - No Linux due to RAM limitation, but there are **µClinux demos** and other OS (Zephyr, NuttX).

� Summary:

- **Real Harvard= no Linux.**
- **Modified Harvard= sometimes µClinux or even real Linux (e.g. Blackfin, TI DSP).**
- **"Harvard on the inside, but unified on the outside" (ARM Cortex-A)= normal Linux.**

Overflow is possible with ARM Cortex-A - just like with practically all modern von Neumann/Modified Harvard architectures.

� Why?

- The **ARM Cortex-A** is a general application processor (e.g. in smartphones, SBCs, routers).
- It supports **32- or 64-bit arithmetic** (ARMv7-A → 32 bit, ARMv8-A → 64 bit).
- If you calculate in a fixed word width (e.g. 32-bit addition) and the result is greater than the maximum value that can be displayed (`0xFFFFFFFFFF` for 32-bit unsigned), an **overflow** occurs.

�Types of overflow

1. **Unsigned overflow**
    - Example: `0xFFFFFFFFFF+ 1→ 0x00000000`
    - The carry flag (`C`) in the program status register is set.
2. **Signed overflow** (two's complement)
    - Example: `INT32_MAX (2147483647)+ 1→ -2147483648`
    - The overflow flag (`V`) is set.
3. **Buffer overflow (software level)**
    - Completely independent of the CPU: if a program writes more data to a memory area than intended, a **buffer overflow** occurs.
    - This is a security vulnerability and a frequent target for attacks (stack smashing, code injection).

�Special features of the ARM Cortex-A

- Has **condition flags** (N, Z, C, V) in `CPSR` (Current Program Status Register) to detect arithmetic overflows.
- In ARMv8-A (AArch64) there are also status flags, so overflow detection is possible on the hardware side.
- Modern Cortex-A cores also have **memory protection (MMU)**, so that buffer overflows can occur but no longer automatically lead to code execution (due to NX bit, ASLR etc.).

�Conclusion:

- **Yes, arithmetic overflows are possible on Cortex-A**, just like on x86 or RISC-V.
- **Software buffer overflows** are also possible, but modern security mechanisms make them more difficult to exploit.

�Stack overflow

- **What is the stack?**
  The stack is the area in RAM that a program uses for **local variables, function calls, return addresses**, etc. It typically grows "downwards" (to smaller addresses). It typically grows "downwards" (to smaller addresses).
- **Stack overflow (classic)**
  → When a program uses more memory on the stack than is permitted. Examples:
    - Endless recursion (`foo()` calls itself an infinite number of times).
    - Local arrays that are too large (`char buf[1_000_000]`).
- **Buffer overflow in the stack**
  → If you write more data to a stack buffer than there is space for, you overwrite **neighboring variables** or even the **return address**.
    - Classic exploit: **Stack Smashing**→ Attacker overwrites return address to execute own code.
    - Modern defense: **Stack Canaries**, **ASLR**, **NX-Bit**.

�Heap overflow

- **What is the heap?**
  The heap is the memory area for **dynamically allocated memory** (e.g. `malloc`, `new`). It grows independently of the stack.
- **Heap overflow**
  → When a program writes more data to a heap buffer than has been allocated. Example:

```
c
char *p= malloc(16);
strcpy(p, "this string is much too long...");
```

  - Now adjacent heap structures are overwritten.
  - Dangerous because the heap contains internal management structures (`malloc metadata`, free lists, function pointers).
- **Exploits**
  - Overwriting heap metadata → Control over memory management.
  - **Use-After-Free**+ Heap overflow= Powerful attack vectors.
  - Today complicated by **heap hardening** (e.g. glibc `malloc checks`, safe linking, ASLR).

�Differences compact

Feature	Stack overflow �	Heap overflow📦
Storage location	Stack (local variables, function frame)	Heap (dynamic allocation)
Typical cause	Recursion, local arrays too large	Incorrect memory management, `malloc/free` error
target	Return address, local variables	Heap metadata, function pointers, other objects
Exploit form	Stack Smashing, ROP	Heap spraying, use-after-free, arbitrary write
Countermeasures	Stack canaries, NX bit, ASLR	Hardened malloc, ASLR, Pointer Encryption

---

�Summary:

- **Stack overflow**= too much/too deep stack usage or buffer overflow on the stack → often return addresses affected.
- **Heap overflow**= Write error in dynamic memory → Often affects management structures or neighboring objects.

If we look at **stack and heap overflows** specifically in the context of **Harvard architectures**, there are a few differences to von Neumann architectures (such as x86 or ARM Cortex-A).

�Special feature of the Harvard architecture

- **Strict separation of instruction memory (flash/ROM) and data memory (RAM).**
- Code and data are **physically separated**, i.e. data overflows **do not** usually overwrite the program code directly.
- This makes classic **code injection via buffer overflow** more difficult - but not impossible.

�Stack overflow in Harvard architectures

- **Where is the stack located?**
  Typically in **RAM** (data memory).
- **Stack overflow classic** (e.g. AVR, PIC, DSP):
  - Endless recursion or too large local variables → Stack "grows" into areas intended for something else (e.g. global variables).
  - On small MCUs without MMU **there are no protection mechanisms** → immediate corruption.
- **Exploit potential:**
  - Overwriting the return address only works if it is also stored in RAM.
  - Many Harvard CPUs store return addresses in a separate **hardware stack** or even in the **program counter stack** → which makes **classic stack smashing** difficult or impossible.
  - Example: **PIC16** has only a tiny hardware stack without RAM access → cannot be overwritten.

📦 Heap overflow in Harvard architectures

- **Heap on MCUs:**
  - Many classic Harvard microcontrollers (AVR, PIC) have **no dynamic heap at all** or only very limited malloc `implementations`.
  - Dynamic memory management is often unusual on small MCUs → little attack surface.
- **Where available** (larger DSPs, Harvard-like MCUs):
  - Heap is located in the **data memory (RAM)**.
  - A heap overflow overwrites neighboring objects or management structures → classically exploitable.
- **But:** Since data and program memory are separate**, the executable code is never** overwritten **directly**. Exploits must run via **indirect manipulation** (e.g. function pointers in RAM).

�Comparison with von Neumann architectures

Characteristic	Von Neumann (e.g. x86, ARM Cortex-A)	Harvard (classic: AVR, PIC, DSP)
Stack overflow	Return address can be overwritten (classic stack smashing)	Return address often difficult to manipulate in hardware stack →
Heap overflow	Management structures+ Possibly code memory can be overwritten	Only data memory affected, code in ROM remains secure
Code injection	Possible (RAM and ROM can be used together)	Directly difficult, as code in separate flash/ROM

Feature	Von Neumann (e.g. x86, ARM Cortex-A)	Harvard (classic: AVR, PIC, DSP)
Exploit vector	Return-Oriented Programming (ROP), shellcode	Data manipulation, logic errors, rarely ROP

�Conclusion:

- **Stack and heap overflows also exist in Harvard architectures**, but their impact is limited because code and data are physically separated.
- **Stack overflow**: often less dangerous if return addresses are located in the non-overwritable hardware stack.
- **Heap overflow**: can still corrupt internal data structures or function pointers, but no direct code overwriting.
- Attacks are possible, but often only via **controlled data corruption** (manipulation of function pointers, state machines, protocol data).

�Example: Stack overflow on AVR (Harvard architecture)

```c
// AVR example (e.g. ATmega328P)
#include <avr/io.h>

void foo(int x) {
    char buf[8];               // small buffer on the stack
    for (int i= 0; i< x; i++) {
        buf[i]= 'A';           // no bounds checks!
    }
}

int main(void) {
    foo(32);                   // much too large, 32> 8
    while (1) {}
}
```

What happens here?

- `buf` is in the **RAM stack**.
- We write 32 bytes, although only 8 are intended.
- Result: we overwrite **neighboring areas on the stack**.

But on AVR:

- Return addresses (program counters) are saved **on the stack in RAM**.
  → A **classic stack smash** is **possible** here, but:
    - Address space is very small (e.g. 64 KB Flash, 2 KB RAM).
    - Instruction memory (flash) **cannot be written to at runtime**.
- Effect: The µC jumps to "garbage addresses"→ Program crashes→ usually **reset by watchdog**.
- Exploit difficult because you cannot reload shellcode (flash≠ RAM).

📦 Example: Heap overflow on AVR

```
c
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *p= malloc(8);
    strcpy(p, "This is too long!"); // writes beyond the end while (1) {}
}
```

What happens here?

- AVR has a minimal `malloc` implementation (heap in RAM).
- The string overwrites **heap metadata** or **neighboring objects**.
- Effect: The next `malloc` or `free` runs into a corruption→ Program crashes.

But:

- Heap is only in RAM→ no possibility to overwrite code directly.
- Again: maximum data manipulation, no classic code injection.

�Why exploits are difficult on Harvard CPUs

- **Separation of code/data**: Flash (instructions) cannot be written to like RAM.
- **Small address spaces**: difficult to build complex ROP chains.
- **Simple systems**: no operating system, no system calls to exploit.

☞ Typical effect: crash or unpredictable behavior.

☞ Exploits are more **logical**:

- Manipulation of function pointers in RAM,
- incorrect states in protocols,
- control of hardware registers through overflow.

�Conclusion using the AVR as an example:

- Stack and heap overflows also occur on Harvard architectures.
- **Exploitability is severely limited** because the code area (flash) cannot be overwritten.
- Usually it only leads to crashes, rarely to full code execution.

⚡ Example: Function pointer manipulation (AVR, Harvard CPU)

```
c
#include <avr/io.h>
#include <string.h>
```

```
#include <stdlib.h>

void led_on(void) {
    PORTB|= (1<< PB5);       // Switch on LED
}

void led_off(void) {
    PORTB &= ~(1<< PB5); // Switch off LED
}

int main(void) {
    DDRB|= (1<< PB5);        // LED pin as output
    void (*fp)(void)= led_off;      // Function pointer points to "LED off"

    char *buf= malloc(8);           // small heap buffer
    // Attacker controls input:
    strcpy(buf, "AAAAAAAAAAAAAAAAAAAAAAAAAA"); // way too long!

    // Somewhere in memory: fp is directly after buf fp();
             // Call the manipulated function pointer while
    (1) {}
}
```

🔍 What happens here?

- The heap buffer `buf` is only 8 bytes in size.
- `strcpy` writes 24 bytes → overwrites the neighboring memory.
- Directly behind this is `fp`, the function pointer.
- The address in `fp` is changed by the overflow.

� Attack vector

- Normally: `fp= led_off`.
- After overflow: `fp` could point to `led_on` (or a completely different function in the flash).
- When calling `fp()`, the AVR jumps to an address set by the attacker in the **instruction memory (flash)**.

🔒 Restrictions on Harvard architecture

- The attacker **cannot inject new code** because the flash cannot be written in the run.
- However, he can **misuse existing code**:
  - Redirect function pointers to other legitimate functions.
  - Combination= primitive form of **Return-Oriented Programming (ROP)**.
- Result: Attacker can **change control flow** even without injecting new instructions.

�Comparison with von-Neumann

aspect	Von Neumann (e.g. x86, ARM-A)	Harvard (e.g. AVR, PIC)
Code injection	Possible (RAM executable)	Not possible (Flash≠ RAM)
Function pointer manipulation	Possible	Possible
ROP	Fully expandable	Only limited (small code pool, simple ISA)

�Conclusion:
Overflows are also **dangerous** on Harvard architectures such as AVR/PIC/DSP - not through code injection, but through **control flow manipulation via function pointers or state machines**.

How "ROP" can basically work on Harvard CPUs

Framework conditions (AVR/PIC/DSP, classic Harvard):

- **Code in Flash/ROM**, **data (stack/heap) in RAM**.
- **No** "Execute code from RAM"→ **No code injection**.
- **But**: return addresses are **in the RAM stack** (with many AVRs/DSPs); function pointers are in **RAM**.
  ⇒**Control flow** can **be redirected** by overflows.

1) Entry point (control flow hijack)

- **Stack overflow** overwrites a **return address** on the stack
  or
- **Heap overflow** overwrites a **function pointer** / vtable pointer.

Result: The next `ret` or indirect function call **does not** jump back to the legitimate location, but to **existing code** in the Flash (e.g. a short library routine).

2) "Gadgets" in existing code

- A **gadget** is a short instruction sequence that ends with a `ret` (or indirect jump).
- On AVRs, these are often the ends of **library functions**, ISRs or compiler logs/epilogs that:
  - Manipulate registers/status,
  - Read/write memory,
  - set I/O registers,
  - and execute `ret` at the end.

Since Flash is not writable, the attacker **only** uses **existing fragments**.

3) Chain formation (return/jump-oriented)

- The attacker uses **prepared stack content** (manipulated return addresses+ possibly parameters expected by Prolog/Epilog/ABI) or **prepared RAM data** to control which gadget is executed next:
  1. **which gadget** is executed next,

2. **which side effects** (e.g. write accesses) occur,
3. **which registers/values** are present when the next gadget starts.

On very small MCUs, the selection of useful gadgets is limited; nevertheless, it is often sufficient to **change states** (e.g. config flags, function pointers, protocol state, I/O pins).

How does this differ from ROP on x86/ARM-A?

Aspect	Von Neumann (x86, ARM-A)	Harvard (AVR, PIC, DSP)
Code injection	Often possible (if W^X is missing)	**Not** possible (Flash ≠ RAM)
Gadget density	High (rich std-libs/ELF/ASLR bypass required)	Low, but available (lib ends, ISRs, trampolines)
Targets	Arbitrary code, syscalls, shell	Mostly **state/data manipulation**, I/O, function redirection
Stability	Higher (more tools)	Often fragile (small memory, watchdog resets)

Defense checklist (practical for Harvard/MCUs)

Compiler & Linker

- Enable **stack protection** (where available): `-fstack-protector-strong` (AVR-GCC supports basic variants), **keep frame pointer** for debugging.
- Use **fortified libs** (safe variants of `memcpy/strcpy`, e.g. length-checking wrappers).
- **Link-Time Optimization (LTO)** & **dead code elimination**: reduces gadget surface.
- **Minimize function pointers** or keep them only in **static** `areas`; convert critical call sites to **direct calls**.

Memory & architecture

- **Lock self-programming** (AVR: set **lock bits** → prevents subsequent flash write commands `SPM`).
- Use **MPU/region protection** if available (for Harvard-like MCUs with MPU/Cortex-M-MPU).
- **Limit/monitor stack size**: Guard pattern (canary in RAM), check periodically.
- **Avoid or encapsulate heap**: fixed buffers, ring buffer; if heap is necessary → **Hardened malloc/Boundary checks**, **quarantine** (Delay-Free) against Use-After-Free.

Coding practice

- **Strict bounds checks** (never strcpy/gets-type APIs).
- Check **explicit length fields** in logs; reject-by-default.
- Leave **constant data in** PROGMEM/Flash (do not mirror in changeable RAM structures if not necessary).
- **Validate indirect calls** (callback tables): Target address whitelist, CRC of tables on boot.

System & runtime

- Configure **watchdog** sensibly (crash → fast fail-safe, but not as a replacement for fixes).
- Activate **brown-out detection** (prevents random memory corruption in the event of voltage drops).
- **Secure boot / image signatures** (where available) to block tampered firmware.

Tests & Analysis

- **Fuzzing** on input interfaces (UART, CAN, RF, USB, ...) with length checks.
- **Address sanitizer** is not available on many MCUs; instead **instrumentation**: red zones around buffers, assertions.
- **Code review**: Prioritize hot paths with `pointers/memcpy`; audit tables for ISR/callback pointers.

Note

> On Harvard MCUs, "ROP" is less "remote shell", more **"controlled state change"**: Overflows are used to bend **control flow** to **existing flash routines** in order to influence **critical data/hardware registers.**
> to influence critical data/hardware registers.

There are no AVR boards or AVR MCUs with MMU (Memory Management Unit).

� Why?

- **AVR architecture** (classic 8-bit AVR, e.g. ATmega/ATtiny, and also the later 32-bit AVR32) is designed for **small embedded systems**.
- Goal: **simplicity, low power consumption, low costs**.
- An **MMU** is comparatively complex (paging, virtual addresses, guard rings ...) → unnecessary overhead for typical AVR use cases (sensors, controllers, Arduino projects).

�What does this mean in practice?

- AVR → always works with **physical addresses** in RAM/flash.
- No **virtual memory**, no **process protection** → a buffer overflow can immediately crash into global variables or the stack.
- Therefore: **no Linux** on AVR (because Linux requires MMU or at least MPU).
- **µClinux** runs on some Harvard-like architectures without MMU (e.g. ARM Cortex-M), but **not on AVR**.

�Comparison: Who has an MMU?

- **ARM Cortex-A (smartphones, SBCs)** → yes, with MMU → Linux runs.
- **ARM Cortex-M (MCUs, STM32, nRF52)** → only an MPU (Memory Protection Unit), no full MMU.
- **RISC-V (larger cores, e.g. SiFive U series)** → yes, MMU possible.
- **AVR / AVR32** → no, never equipped with MMU.

�Conclusion:
AVR boards (Arduino, ATmega, ATtiny, even the old AVR32 chips) **do not have an MMU**. This keeps them in the **bare-metal / RTOS** area, but **Linux** or process isolation are excluded.

�AVR (classic Harvard, no MMU)

- **Separate memories**: Flash (instructions) and RAM (data).
- **Advantage**: Code cannot simply be overwritten by a buffer overflow → certain "built-in security".
- **Disadvantage**:
  - No MMU → no virtual memory, no process protection, no memory protection.
  - An overflow in RAM immediately corrupts neighboring variables or function pointers.
  - No Linux, only bare-metal or RTOS.

�RISC-V (mostly von-Neumann, with MMU option)

- **Uniform memory**: instructions and data in the same RAM/ROM area.
- **With MMU** (e.g. RV64GC, SiFive U series, Linux-capable):
  - Virtual memory, processes isolated from each other.
  - "Great" protection against classic overflows (segfault instead of silent corruption).
  - BUT: Code and data share the memory → Code injection possible in principle (except prevented by W^X/NX bit).
- **Without MMU** (e.g. small RV32 controller):
  - Behavior as with Cortex-M or AVR → everything "naked" on physical memory.

�ARM Cortex-M (Modified Harvard, MPU only)

- Harvard-like: separate buses for data/instructions, but uniform address space.
- **MPU instead of MMU**:
  - Protection areas can be defined (e.g. stack read-only, flash non-writable).
  - No virtual memory management, no multitasking as with Linux.
- → Security lies between AVR (no protection at all) and "real MMU system" (RISC-V with Linux or ARM Cortex-A).

�Summary (pictorial)

Architecture	Harvard separation	MMU	Practical consequence
AVR	�Strict	�No	No code injection, but no isolation - everything in RAM insecure
RISC-V (with MMU)	�von-Neumann	�Yes	Linux-compatible, process protection, but susceptible to code injection without NX/W^X
Cortex-M (with MPU)	�"Modified Harvard"	⚠�MPU (partial protection)	Limited protection, no real Linux

Architecture	Harvard separation	MMU	Practical consequence
Cortex-A (smartphones, SBCs)	�Modified Harvard (caches)	�Yes	Full-fledged OS, strong protection mechanisms

---

�So yes:

- With **Harvard** (AVR) you lose the "big" memory isolation (MMU),
- With **MMU** (RISC-V/ARM-A) you lose the hard physical separation of code/data.

But: Modern systems (RISC-V, ARM-A) **replace Harvard security** with software/hardware policies:

- **NX-Bit / W^X** → code and data must not both be "executable+ writable".
- **ASLR, Stack Canaries, Fortified Libs** → make overflows much more difficult.

�A comparison of security mechanisms

Architecture	Harvard separation	MMU	MPU	NX-Bit / W^X	ASLR	stack
AVR (ATmega, ATtiny)	�Strict (code in flash, data in RAM)	�	�	�	�	�
AVR32 (set)	�Strict	�	�	�	�	�
ARM Cortex-M (M0-M7)	�Modified Harvard (separate buses, standardized addresses)	�	⚠�Yes, MPU (regions only)	⚠�Partially (Flash read-only, RAM exec)	�	�(rarely in
ARM Cortex-R	�Modified Harvard	�	�MPU	⚠�Partially	�	�
ARM Cortex-A (smartphones, SBCs)	�Modified Harvard (caches, separate I/D paths)	�	-	�NX-Bit/W^X	�	�
RISC-V small (RV32-MCUs)	�von-Neumann	�	⚠�optional MPU	�(except vendor HW)	�	�
RISC-V large (RV64GC, Linux-capable)	�von-Neumann	�	-	�NX-Bit/W^X	�	�
DSPs (TI C6000, Blackfin)	�Modified Harvard	⚠�partly rudimentary	�⚠�	�	�	�

�Briefly explained

- **Harvard separation**: makes code injection more difficult because instruction flash is not in RAM.
- **MMU**: full virtualization → Process isolation, paging, Linux-capable.
- **MPU**: light memory protection (regions writable/executable or not).
- **NX-Bit / W^X**: prevents memory areas from being writable and executable at the same time (critical against code injection).
- **ASLR**: Address space randomization makes ROP/JOP exploits more difficult.
- **Stack canaries**: Markers in the stack to detect overflows in good time.

�Conclusion

- **AVR**= "simple, no protection mechanisms, but flash separated → no classic code injection, only data corruption."
- **Cortex-M**= a bit more secure (MPU), but no full protection.
- **Cortex-A / RISC-V Linux**= full security features as on PCs.
- **DSPs / exotics**= mostly without protection, focus on performance.

Waver

Manufacturers who offer small quantities in a sensible way

UniversityWafer, Inc.

- **Offers** high-quality silicon wafers and other substrates **starting from one piece**.
- Fast shipping (often within 24 hours), good prices and flexible stock.Micross+14University Wafer+14WaferPro+14University Wafer+2SPI Supplies+2University Wafer+1

MSE Supplies

- Sells silicon wafers, also with special specifications.
- Popular with research and universities - sometimes with flexible smaller minimum order quantities.ACTIVE BUSINESS COMPANY GMBH

Si-Mat (Germany)

- Offers silicon wafers in almost any desired specification.
- **Small quantities** are also possible - ideal if you have individual requirements.WaferPro+15Silicon Materials+15University Wafer+15WaferPro+3Silicon Wafer Buy+3Wafer World+3

Epiel (Epitaxial wafer specialist)

- Specialized manufacturer of silicon epitaxial wafers (also on wafer base).
- Supplies in **customized small quantities** - e.g. epi-layers according to your specifications (3" to 8") epi wafers.

Short comparison

manufacturer	Minimum quantity	Special features
UniversityWafer	From 1 wafer	Extensive stock, fast delivery
MSE Supplies	Small quantities possible	Scientific focus, tailor-made specifications
Si-Mat	Also individual items	Customized solutions from Germany
Epiel	Small series possible	High-quality epitaxial wafers according to customer requirements

---

Product recommendation (typical for consumers / hobby users)

Entry-level size - immediately available, good

price 4-inch N-Type Silicon Wafer (CZ, 1-10 fi ·

cm) € 23.95

Soka Technology

Compact size with extra oxide layer for experimental purposes 3-

inch P-type silicone wafer with oxide layer

29,95 €

Soka Technology

ChatGPT selects products independently. Learn more

- **4-inch N-Type Silicon Wafer (CZ, 1-10 Ω· cm)**
  High-quality silicon wafer, ideal for experiments or prototyping - directly available with global shipping.
- **3-inch P-type silicon wafer with oxide layer**
  Suitable for smaller projects - comes with an oxide layer and is also already listed in the store.

If you are looking for **single wafers** for hobby, research or smaller projects, **UniversityWafer**, **MSE Supplies**, **Si-Mat** and **Epiel** are great places to start.
If you want to compare directly and value easy availability, the **4″ N-Type wafer** or the **3″ P-Type wafer with oxide layer** is a practical and affordable option.

You *could* **build a Harvard CPU with an MMU - but this is uncommon and would have interesting consequences.**

�Why unusual?

- **MMU (Memory Management Unit)** works with **virtual addresses** that are mapped to **physical addresses**.
- In classic **Harvard CPUs**, however, there are **two (or more) physical memory worlds**:
  - Instruction memory (Flash/ROM)
  - Data memory (RAM)
- An MMU would therefore have to manage **two separate mapping tables**: one for **instruction fetches**, one for **data accesses**.
- This makes the hardware more complex, without the typical target group of Harvard architectures (microcontrollers, DSPs) benefiting much from it.

�Theoretically feasible

There are architectures that already go in this direction:

- **ARM Cortex-M with MPU**: no real paging, but you can define **regions**→ "light MMU". Modified Harvard (separate buses, but common address space).
- **DSPs (TI C6000, Analog Devices Blackfin)**: Modified Harvard, some with protection mechanisms and even Linux ports.
- **ARM Cortex-A**: internally Harvard-like (separate instruction/data caches), **but** externally like a von Neumann system. A "classic" MMU sits in front of it→ and yes, Linux runs. So: **The concept already exists in hybrid form.**

�If you were to "build it yourself"

Imagine this:

- You design a CPU with **separate instruction and data buses** (Harvard).
- In front of this you place an MMU that has **two translation lookaside buffers (TLBs):**
  - I-TLB for instructions
  - D-TLB for data
- This is exactly what an **ARM Cortex-A or modern RISC-V** already does in principle:
  - Internal Harvard (separate caches & buses)
  - MMU in front→ uniform, virtual address space

☞ So you could say that **an ARM Cortex-A** *is in fact* **a "Harvard CPU with MMU"**, only implemented as a modified Harvard.

�Conclusion

- **Classic AVR/PIC→ no MMU, because too simple.**
- **Theoretically→ yes, Harvard+ MMU works**, you would have to have two parallel mapping structures.
- **Practically→ modern high-end CPUs (ARM-A, RISC-V, x86)** are already *hybrid Harvard* with MMU.

�Why you *don't* usually see a "pure Harvard CPU with MMU"

1. **Two address spaces→ two translation tables**
   - A Harvard core fetches **instructions** from one memory space (Flash/ROM) and **data** from another (RAM).
   - An MMU needs to translate **every memory access** (virtual→ physical).
   - That means you'd need **two MMUs (or one MMU with dual TLBs)**: one for instruction fetches, one for data.
   - More silicon area, more complexity, more power draw→ not attractive for the typical use cases of Harvard cores (tiny, low-cost MCUs).
2. **The benefit doesn't match the cost**
   - The whole point of an MMU: isolation, paging, process separation, demand paging for OSes like Linux.
   - The whole point of Harvard: **simplicity and speed** in small controllers, DSPs, and real-time systems.
   - If you add an MMU, you lose much of the "simplicity and determinism" advantage of Harvard.
3. **Caches already solve the performance problem**
   - Modern CPUs (ARM Cortex-A, RISC-V, x86) are internally *modified Harvard*:
     - Separate instruction & data caches for performance.
     - A **unified address space with an MMU** for OS features.
   - This gives the best of both worlds→ you don't need a *strict* Harvard MMU design.

�But could it exist?

Yes, and in fact **it already kind of does**:

- **ARM Cortex-A**: internally Harvard (split I-cache and D-cache), externally unified, with a full MMU.
- **Some DSPs (TI C6000, Blackfin)**: Harvard-style pipelines, with memory protection features.
- These are essentially "Harvard cores+ MMU," just not marketed that way.

�Why nobody builds a "pure Harvard+ MMU" chip

- Market mismatch: Harvard is used in **small MCUs** → too tiny to justify MMU.
- Big CPUs already use a **hybrid (Modified Harvard+ MMU)** because it's more practical for general-purpose computing.

So the real answer is: **we don't see it, not because it's impossible, but because the two design goals (Harvard= simple/embedded, MMU= complex/OS) almost never overlap.**

�TL;DR:

It's not a fundamental impossibility. It's just that:

- **MCUs (Harvard)** don't need MMUs,
- **Big CPUs (with MMU)** already use Modified Harvard internally.

�Harvard+ MMU vs. Modified Harvard+ MMU

Feature	Pure Harvard+ MMU (theoretical)	Modified Harvard+ MMU (practical, e.g. ARM Cortex-A)
Address spaces	Separate: instructions (flash), data (RAM)	Uniform: one virtual address space for code & data
MMU design	Two MMUs or dual TLBs required (I-TLB & D-TLB independent)	One central MMU is sufficient, generates unified VA → PA mapping
Caches	Instruction cache+ Data cache, but completely separate address spaces	Instruction cache+ Data cache, both access the same phys. memory
Code injection	Very difficult: code memory (flash) cannot be written in RAM	Possible, but prevented by NX-Bit/W^X
Overflow exploits	Data overflows only corrupt RAM, not code	Data overflows can theoretically overwrite code, but MMU/NX blocks
Flexibility	Low: Programs must observe separate instruction and data spaces	High: uniform address space, dynamic loading, shared libraries
Complexity	Higher (manage two MMU paths)	Low (one MMU path, then split caches)
Area of application	Could be interesting for safety-critical MCUs (safety/automotive), but rarely built	General CPUs (smartphones, servers, Linux-enabled systems)
In practice	Hardly realized - AVR/PIC have no MMU, DSPs only protection registers	Standard in ARM Cortex-A, RISC-V Linux cores, x86

�Conclusion

- **Harvard+MMU**:
  - Theoretically more protection against classic code injection (instruction memory≠ data memory).

    - Practically unattractive because software and compilers would have to be much less flexible and hardware would become more complex.
- **Modified Harvard+MMU (common today)**:
    - Uniform memory model for software (simpler for OS & compiler).
    - Nevertheless, high performance due to separate caches and buses.
    - Attacks by overflow→ fended off by modern security features (NX-Bit, ASLR, Canaries).

�Scenario: Harvard+MMU in server applications

1) Motivation

- In classic server CPUs (x86, ARMv8, RISC-V Linux), **code and data are located in the same physical memory**.
- Protection mechanisms such as **NX-Bit (Write XOR Execute)**, **ASLR** and **Stack Canaries** reduce exploit risks, but:
    - **JIT engines** (Java, JavaScript, Python) must dynamically use memory areas sometimes as data, sometimes as code. → Attack surface.
    - A **memory safety bug** (buffer overflow, use-after-free) can still lead to code execution if protection is bypassed.

☞ A **real Harvard+MMU system** could be one level more secure here, as **instruction memory** remains **physically separated**.

---

2) Architectural idea

- **Harvard structure**:
    - **Instruction memory (code)** is located in **read-only flash/NVRAM/ROM** (or dedicated DRAM channel, read-only).
    - **Data memory (RAM)** remains writable.
- **Dual MMU**:
    - **I-MMU (instruction side)**: maps virtual code addresses to physical code pages.
    - **D-MMU (data side)**: maps virtual data addresses to physical RAM pages.
- **Server OS (Linux/BSD)** would have to be adapted:
    - Dynamic reloading of code (shared libs, JIT) would mean: Copying data→ code memory via a **secure transfer mechanism** (e.g. only by kernel).
    - No app could ever have direct "write" access to its code area.

---

3) Advantages in server use

- **Absolute code integrity**:
    - No exploit can overwrite instruction memory (even if the root user causes memory corruption).
    - Malware that wants to inject shellcode **always fails**.
- **Strict policy enforcement**:
    - Code pages= are immutable.

  - Only signed binaries may be loaded into the instruction memory (secure loader).
- **Attack surface reduction**:
  - Classic buffer overflow exploits → only data corruption, no code injection.
  - ROP/JOP attacks more difficult, as only a limited code area is available in the instruction memory.

4) Disadvantages & challenges

- **JIT/interpreter performance**:
  - JITs such as in Java, JavaScript, .NET would have to load their code image into the instruction memory via kernel APIs → Performance loss.
  - Alternative: "data+code cache" in hardware (complicated).
- **Complexity of the MMU**:
  - Two MMUs= double the management effort.
  - Operating system would have to manage two address spaces synchronously (instructions vs. data).
- **Flexibility vs. security**:
  - Servers with a lot of dynamic code (JIT, WebAssembly, VM) suffer.
  - More suitable for **high-security servers** where absolute code integrity is more important than performance.

�Practical scenario

☞ Imagine a **banking or military server:**

- Runs a customized Harvard+MMU CPU.
- **OS Kernel+ Hypervisor** are *read-only* in the instruction memory.
- Applications may only run in a sandbox whose code only enters the instruction memory via a **signed loader**.
- **Exploit from data (buffer overflow, SQL injection)** → destroys data, but **never code flow**.
- Result:
  - Extremely high resistance to **remote code execution**.
  - Very interesting for **zero trust servers** or **critical infrastructure**.

�Conclusion:

A **Harvard+MMU server** would be a "code execution-proof server" in practice -

- Very robust against exploits,
- but inflexible for JIT/VM-based workloads,
- only useful in **high-security niches** (banks, military, secret services, crypto servers).

�Possibilities

1. MPW (Multi Project Wafer) Services

- Several customers share a wafer (costs are split).
- You only pay for "your space" on the wafer.
- Provider:
  - **MOSIS (USA)**→ mosis.com
  - **Europractice (EU, mainly universities & SMEs)**→ europractice-ic.com
  - **efabless** with the **Google/SkyWater PDK** (130nm open source)→ efabless.com
- Prices: from a few thousand$ for a small test design.
- Result: you get some **manufactured chips** back, not a whole wafer.

2. MPW Shuttle via Foundries

- **TSMC, GlobalFoundries, TowerJazz, SkyWater** offer MPW shuttles (mostly via resellers such as MOSIS/Europractice).
- Advantage: Professional processes (28nm, 65nm, 130nm etc.).
- Disadvantage: Access often only via universities, companies or consortia.

3. Commercial wafer production (own run)

- You can of course also **"book" entire wafers** from foundries such as TSMC, GlobalFoundries or SMIC.
- But it costs **millions** (mask set+ production run).
- Rather only realistic for large companies.

4. Research & maker-friendly

- **TinyTapeout**: great start, open-source hardware design, you get your logic block in an MPW run on SkyWater 130nm.
  - Website: tinytapeout.com
  - Costs: sometimes< 500€ for a small module.
- **Efabless/Google SkyWater Shuttle**: free/sponsored if your design is open source.

�Conclusion

- **For hobby / prototyping**:
  - **Efabless / TinyTapeout / Europractice MPW**= ideal, cheap, realistic.
- **For companies / serious projects**:
  - **MOSIS / Foundry MPW Shuttle**.
- **Own waveruns**: theoretically yes, but very expensive (>> €1 million).

�Entry options for hobbyists & small teams

1. TinyTapeout

- **What:** Open community project (SkyWater 130nm), you book a small space on a multi-project wafer (MPW).
- **Costs:** from approx. 200-500 €.
- **Output:** You get back some finished chips on an eval board.
- **Special feature:** Extremely low entry hurdle, good community.
- 🌍 tinytapeout.com

2. Efabless / Google SkyWater Shuttle

- **What:** Collaboration with Google, SkyWater 130nm PDK.
- **Cost: Free** if your design is open source (GPL, Apache, etc.).
- **Output:** A few dozen chips will be sent to you.
- **Special feature:** Full shuttle run, but only for open source.
- 🌍 efabless.com

3. Europractice (EU)

- **What:** Access to MPW shuttles from TSMC, GlobalFoundries, ON Semi.
- **Cost:**
    - NRE (masks + space) from approx. 5000 €.
    - Chips depending on process and quantity.
- **Special feature:** Universities and SMEs get easier access, hobbyists via cooperation with a university.
- 🌍 europractice-ic.com

4. MOSIS (USA)

- **What:** A veteran of MPW shuttles, offers processes from SkyWater to TSMC.
- **Costs:** similar to Europractice, from a few thousand dollars.
- **Special feature:** Very established, lots of support, but rather academic/commercial.
- 🌍 mosis.com

5. Multi-Project Wafer Runs (Foundries direct)

- **What:** TSMC, GlobalFoundries, TowerJazz offer their own MPW shuttles.
- **Costs:** from 10-20k€ for very simple designs (65-180 nm).
- **Special feature:** Access mostly via resellers (MOSIS, Europractice).

�Costs & expectation

- **Cheap & maker-friendly:** TinyTapeout (200-500 €)→ perfect for first experiments.
- **Academic/startup:** Efabless Shuttle (free with open source)→ great for small teams.
- **Semi-professional:** Europractice / MOSIS (5k-20k €)→ more control, more modern processes.
- **High-end:** Own wafer run (100k-1M €)→ only for companies.

�Tip:

If you really want to make a **Harvard+MMU CPU on a small scale**, **TinyTapeout** or **Efabless Shuttle** would be the perfect first step. You could describe a minimal CPU in Verilog, submit it, and after a few months a few months you could have chips on Eval boards in your hands.

What would be a realistic clock speed at 130nm?

ChatGPT:

The **realistic clock frequency depends** heavily on **the complexity of the CPU** and the **objectives**.

�Typical clock frequencies at 130 nm

- **Simple microcontroller designs (8/16 bit, Harvard architecture, no cache, no deep pipeline):**

  ☞ **20-80 MHz** (very energy-saving, robust).

  - Example: early AVR or MSP430-like designs.
- **Simple 32-bit CPUs without a deep pipeline (e.g. OpenRISC, early ARM7/ARM9):**

  ☞ **100-200 MHz** possible.

  - ARM9 in 130 nm was realistically at ~150 MHz.
- **Pipelined RISC CPUs (5-7 stages, simple caches, MMU light):**

  ☞ **200-400 MHz**.

  - Example: ARM Cortex-M3 in 130 nm could achieve ~250-300 MHz.
- **High-end designs (deep pipelines, out-of-order, multiple ALUs, optimized layout):**

  ☞ **500-800 MHz** (but only with heavy hand optimization and special library).

  - Example: Intel Pentium 4 Willamette (180 nm→ 1.5 GHz, at 130 nm later ~3 GHz - but with huge power consumption and cooling).

�For a Maker/TinyTapeout project (SkyWater 130 nm)

- Toolchain (OpenLANE+ SkyWater PDK) is **not as aggressively optimized** as Intel/ARM industrial flow.
- Realistic: **20-50 MHz** stable, without timing voodoo.
- With a lot of optimization and clever layout: perhaps **80-100 MHz**.

�Conclusion

- For a **Harvard+MMU CPU as a hobby project** on SkyWater 130 nm:
  👉 **20-50 MHz**= safely achievable.
  👉 **100-200 MHz**= possible if you optimize aggressively and run the toolchain "at the limit".
- Industrial designs at 130 nm could reach ~500-800 MHz, but this is hardly feasible for makers.

Roadmap

1) Clean up architecture & RTL

- **Harvard+ 5-stage pipeline** (IF/ID/EX/MEM/WB) instead of single-cycle/2-stages.
- **Separate adders**: `PC+4` and branch target do not multiplex → no huge MUX trees.
- **Branches early** (ID/EX) and simplified prediction (static/BTB light) to avoid long Compare → PC path.
- **Load-use interlock**: 1 cycle stall is ok; timing more important than zero-bubble ideal.

2) Shorten critical paths

- Typical culprits: `Regfile read→ Bypass/MUX→ ALU→ Regfile write`, **multiply/divide**, **branch compare**, **cache hit path**.
- **ALU**: split wide MUXes (staggering), reduce fanout (signal duplication).
- **Mul/Div**: multi-stage pipelines (e.g. Booth/Wallace), or as a **multicycle unit** with `start/done`.
- **Comparator** (Branch): independent short path, do not cascade through ALU-MUX.

3) Think memory clean

- **SRAM macros** (instruction/data separated). For 130 nm: plan 1-cycle access; no combinational return path.
- **Scratch pads** instead of "real" caches for the first 100 MHz mark; later small caches (e.g. 1-2 KB) with **fully synchronous** access.
- **Alignment & width**: Select bus width so that one instruction fetch per clock is guaranteed (e.g. 32 bit aligned).

4) Clock & Resets

- A **single clock domain** for the core. **Release** asynchronous resets **synchronously**.
- Avoid **clock gating** early (only after timing closure), otherwise you will generate hard-to-fix hold errors.

5) Synthesis setup (Yosys/OpenROAD or similar)

- **SDC**: `create_clock -period 10ns` (100 MHz), plus `set_input_delay/set_output_delay`
- **Multicycle** for deliberately multicycle paths (e.g. divider), **false paths** for debug/JTAG.
- **Cell libraries**: mixed linking (e.g. _HD+ _HS); **critical nets** may receive HS/Low-Vt.
- **Constraints** on fanout/transition so that synthesis buffers/upsizes in time.

6) Floorplanning & placement

- **Short paths**: cluster regfile ↔ ALU ↔ bypass locally; instruction SRAM close to IF logic, data SRAM close to MEM stage.
- Early **power grid** (stretchy), sufficient **decaps** so that upsizing does not fail later due to IR drop.
- **CTS**: balanced H-tree, keep target skew small; then plan **hold fixes**.

7) Timing closure (STA)

- Work against **worst-corner** (slow/low-V/high-T), then check best-corner for **hold**.
- Iteration: (a) path report → (b) targeted upsizing/buffering/retiming → (c) re-place/route.
- **ECOn** targeted: individual gates larger, segment MUXes, duplicate registers (fanout split).
- **Wires** count: at 130 nm, wire delay is already relevant → Keep hard nets short, shielding if necessary.

8) Secure functionality

- RTL sims+ **Gate-level sims with SDF** (at least smoke tests IF/EX/MEM).
- Test hazard unit (forwarding/stalls) intensively; formal checks for CDC (should not exist) & deadlocks.

9) Power, EM/IR & DRC/LVS

- After timing: insert **clock gating** (IF stage, Mul/Div, caches) → Re-STA (Hold!).
- Check **IR drop** for hotspots (ALU/Regfile/Bypass); if necessary, add more PG metal/straps.
- Final: **DRC/LVS clean**, include antenna fixes (can change timing minimally).

10) Gradual clock increase

- **Milestones**: 25 MHz → 50 MHz → 80 MHz → 100 MHz.
  After each step: STA @ slow-corner, GL-SDF-Smoke, short PPA review (Power/Area).
- If stuck at 80 MHz: first look at **Mul/Div**, **bypass MUX**, **branch path**, **I-SRAM** return **path**.

11) Useful micro-optimizations

- **Bypass nets** selectively: do not provide everything everywhere.

- Keep **PC path** independent (own adder), no "all-through-one-MUX" philosophy.
- **Register file**: 2R1W is sufficient; timing-friendly implementation (hard macro, short MUX depth).
- **Instruction decoder** in steps (Predecode→ Control) to shorten cones.

12) Libraries & trade-offs

- For 130 nm open-flows: start with **HD-Lib**, switch critical paths to **HS-Lib** (Leakage ↑, Fmax ↑).
- If you just miss 100 MHz: **increase VDD slightly** (if the process provides for this) and keep an eye on **thermal**.

Quick checklist (if it hangs)

- Critical path= **ALU-Bypass-MUX**? Segment→ , split fanout, HS cells.
- Critical path = **Mul/Div**? → more pipeline stages or multicyclic + multicycle constraints.
- Critical path= **Branch**? → Compare earlier, separate PC adder, keep target MUX short.
- Critical path= **SRAM hit**?→ fully synchronous, no combinational readback; placement denser.

Why 4 GHz at 130 nm is unrealistic

- **Critical path**: fmax≈1/tpd,criticalf_{max}\approx 1/t_{pd,critical}fmax≈1/tpd,critical. Gate and **wire delays** dominate at 130 nm; global networks, cache accesses, bypass MUXes etc. are very limiting.
- **Deep pipeline required**: 4 GHz⇒250 ps clock period. This means you would have to shrink all critical paths to a few very small logic stages+ extremely short wires. In practice this means: special cells, hand layout, brutal pipelining - and even then **interconnect + clock skew** kill the gain.
- **Clock tree/skew/jitter**: With a 250 hp budget, skew alone quickly becomes a showstopper.
- **Power/IR/EMI**: High Vdd+ Low-Vt cells⇒**Leakage currents & heat** explode; EM/IR drop limited.
- **Tooling reality**: with open-flows/standard cells in 130 nm, **20-100 MHz** (maker setup) to **~200-400 MHz** (very well optimized, small caches) are realistic. Even heavily optimized industrial designs in 130 nm are typically **well below 1 GHz** without massive special sauce.

What SMP really brings

- **More cores≠ higher clock rate**: The OS does not "standardize" computing power to a faster core.
- **Amdahl's Law**: Speedup is limited by the serial part. Good scaling only with highly parallelizable workloads.

Realistic ways to more performance (130 nm)

- **Scale throughput**: 2-8 simple cores @ 50-150 MHz, divide work.
- **Pipeline & memory**: 5-7 stages, synchronous SRAMs, small I/D scratchpad instead of "real" caches, shorten critical paths.
- **Low-Vt/HS cells selectively**+ clean floorplanning (short paths around ALU/regfile).
- **DVFS** and moderate Vdd tuning (in the process window) - pay attention to thermals.

If you want to get into the GHz region

- **Smaller nodes** (90/65 nm and below) or **very** deep, hand-tuned pipelines with a large PPA budget.

- For prototyping: modern **FPGAs** typically manage **300-600 MHz** in hotspots (different construction site, but realistically achievable).

Conclusion: SMP+ own OS helps with **throughput**, not with **clock speed**. In 130 nm, 4 GHz is practically unachievable-plane with tens to a few hundred MHz per core and scale via parallelism.

You:
that's why you have your own os with your own software, which gets to 4ghz via parallelization with smp... i wasn't talking about real 4ghz.

ChatGPT:
Ah, gotcha - you mean **"4 GHz equivalent processing power"** through **SMP parallelization**, not true 4 GHz clock speed. That's realistic **if** workloads parallelize well and the platform is optimized for it.

How many cores do you need?

Target:~ **4 Gops/s** (as a placeholder for "4 GHz at ~1 IPC").

Formula:
Cores NNN≈ 4· 109 ops/sf_core· IPC\dfrac{4\cdot 10^9\ \text{ops/s}}{f_{\text{core}}\cdot \text{IPC}}f_core· IPC4· 109 ops/s Examples

(cleanly calculated):

- **50 MHz**, IPC= 1.0 → 50 Mops/s per core → 4e9/50e6=804e9 / 50e6 = 804e9/50e6=80⇒ **80 cores**
- **80 MHz**, IPC = 0.9 → 80e6· 0.9=72e680e6-0.9= 72e680e6· 0.9=72e6 → 4e9/72e6≈55.564e9/72e6≈ 55.564e9/72e6≈55.56⇒ **56 cores**
- **100 MHz**, IPC= 1.0 → 4e9/100e6=404e9/100e6 = 404e9/100e6=40⇒ **40 cores**
- **150 MHz**, IPC = 0.8 → 150e6· 0.8=120e6150e6-0.8 = 120e6150e6· 0.8=120e6 → 4e9/120e6≈33.334e9/120e6≈ 33.334e9/120e6≈33.33⇒ **34 cores**

Note: This delivers **throughput** like a 4 GHz single core at ~1 IPC - **not the latency**.

When will that work?

- **"Embarrassingly parallel"**: rendering tiles, batch crypto, image/signal blocks, Monte Carlo, map/filter/reduce, search trees with low dependency.
- **High compute-to-data ratio** (compute intensity) so that memory is not limited.

Major stumbling blocks (and countermeasures)

1. **Memory bandwidth**
    - Rough rule of thumb: If ~8 B from the main memory were required per operation, you would need 4e9-8 B/s= **32 GB/s** - illusory at 130 nm.
    - **Antidote:** Scratchpads/L1 reuse, tiling/blocking, streaming DMA, prefetching, SoA layouts.
2. **Cache coherence & scaling**
    - 40-80 cores+ Snoop coherence= expensive.
    - **Antidote: Message passing** (lock-free), **scratch pads** instead of "real" caches, **directory-based** or **non-coherent** caches with explicit flush/invalidate.
3. **Amdahl/Gustafson**
    - Serial share limits speedup.
    - **Antidote:** Switch algorithms to **task graph**, work stealing, avoid fine-grained tasks (batching).
4. **Scheduling & OS overhead**
    - **Context switches** kill throughput at low clock rates.

  - **Antidote: Library OS/microkernel**, per-core run-to-completion, **pinned threads**, zero-copy queues, IRQ steering.

5. **Interconnect**
   - Ring scales poorly over ~8-16 cores.
   - **Antidote: 2D mesh/NoC**, short hop distances, traffic classes (Req/Resp separated).

OS/software design (short & crisp)

- **Model:** SPMD / Actor / Bulk-Synchronous (BSP) with fixed "tiles".
- **Communication:** lock-free MPSC/MPMC queues, fixed mailbox addresses; no global locks.
- **Memory:** per-core **scratchpad**+ explicit DMA; data partitions "owner computes".
- **Scheduler:** per-core run queue, **work stealing** only in case of load imbalance; NUMA-aware.
- **Tooling**: Offload APIs (jobs+ descriptors), fixed batch size, **deterministic** slices.

Realistic target configurations @ 130 nm

- **Variant A:** 40 cores× 100 MHz× ~1 IPC→ ~4 Gops/s
- **Variant B:** 56 cores× 80 MHz× ~0.9 IPC→ ~4 Gops/s
- **Variant C:** 80 cores× 50 MHz× ~1 IPC→ ~4 Gops/s
  - For A/B/C in each case: **Scratchpads+ DMA+ Mesh-NoC+ Message passing**.

Practical checklist

- Split hot loop into **tiles**, keep data local (blocking).
- **Queues** instead of locks; fixed packet formats.
- **Pinned** worker per core; no preemption in the fast path.
- **Batching** of jobs (reduces scheduling overhead).
- Stream **DRAM accesses**, write combining; large bursts.
- **Profiling**: miss rate, bandwidth/core, queue depth, NoC hops.

0) Summary (scroll down for details)

- **Hobby/beginner (TinyTapeout/Sky130, very small designs): €200-€500** (TinyTapeout) or **~€0** (Efabless-Shuttle, *only open-source* if accepted)+ **€200-€1 000** Bring-up (boards, measuring equipment)→ **€400-€1 500** all-in.
- **"Serious" 130 nm prototype (MPW, own packaging, test PCB): €15 000-€60 000** realistic; more ambitious (larger area, better packaging/testing): **€60 000-€150 000**.
- **Fully own wafer run (no MPW):> €1 000 000** (mask set+ production)→ for startups/industry, not makers.

---

1) Minimal path - community/beginners

A) TinyTapeout (SkyWater 130 nm, mini-slot)

- **Shuttle slot:** €200-€500
- **Return delivery:** usually a few chips+ possibly eval board
- **Bring-up/Extras:** €200-€1 000 (test probes, power supplies, possibly own test PCB)
- **Total: ~€400-€1 500**
- **Limitation:** Very small logic area; no "64-core" - rather *proof of silicon*.

B) Efabless/Google Open MPW (Sky130)

- **Shuttle fee: €0** (if open source & accepted)
- **Return delivery:** dozen(s) dies/chips
- **Additional costs (documentation, bring-up, boards):** €500-€2 000
- **Total: ~€500-€2 000**
- **Risk:** Acceptance window/qualification; open source obligation.

---

2) Realistic 130 nm MPW prototype (own chip with packaging)

Assumption: 130 nm MPW (e.g. SkyWater 130, GF/TSMC 130/180 via MOSIS/Europractice), **die area 10-25 mm²**, return delivery **~20-100 dies**, of which **10-50 are packaged**.

Fixed/variable items (strip widths)

1. **MPW shuttle participation (NRE share)**
   - **€8 000 - €30 000** (depending on foundry, area, shuttle provider)
   - Larger area/modern options→ rather **€30k-€60k**
2. **Packaging (standard QFN/QFP/BGA, mold, not ceramic)**
   - Small series 20-50 pcs: **€20-€60 / chip**
   - One-time **setup/handling**: **€500-€2 000**
   - **Example:** 40 pcs × €40 + €1 000 setup → **€2 600**
3. **Wafer probing / sort (optional, highly recommended)**
   - **€1 500 - €6 000** (simple Parametric/Go-No-Go on MPW dies)
4. **Test-PCB / Eval-Board**
   - 2-4 layers, assembly, socket/adapter: **€800-€2 500** per board
   - 2-3 iterations (error/risk buffer): **€1 500-€5 000** total
5. **Power supplies / measuring equipment / lab stuff**
   - Power supplies, level adapters, fixtures, connectors: **€300-€1 500**

6. **EDA / implementation**
    - **Open source flow** (OpenLane/OpenROAD): **€0** license costs
    - Commercial tools (Synopsys/Cadence): **high**→ usually not necessary for first-time MPW users
7. **Engineering time (optional in €)**
    - RTL, verification, PnR, DfT, doc: **Person-months**.
    - If valued: **€10 000-€50 000** (1-3 PM @ market daily rates)

Examples of totals

- **Slim MPW proto (10-12 mm² die, 20-30 packaged chips):**
    - MPW €12 000
    - Packaging €1 800
    - Sample €2 000
    - Test PCB €1 500
    - Lab stuff €700
    - **Subtotal (without dev time): ~€18 000**
    - **With buffer: €20 000-€25 000**
- **Ambitious proto (25 mm², 40-50 packaged chips, more tests):**
    - MPW €25 000-€45 000
    - Packaging €2 500-€4 000
    - Sample €3 000-€6 000
    - Test PCB €3 000-€5 000
    - Lab stuff €1 000-€1 500
    - **Subtotal (without dev time): ~€35 000-€60 000**
    - **With buffer: €40 000-€70 000**
- **With priced-in engineering time (2-3 PM)+ Risk buffer:**
    - **€60 000-€120 000** (realistic for "own chip, real bring-up chance")

---

3) Especially for your 64-core concept (8×8 tiles @ Sky130)

Recommendation:

1. **First silicon slimmed down** (e.g. 4×4 tiles, small SPMs, simple NoC-FIFOs)→ **~10-15 mm²**
2. **Second spin** on 8×8, after timing/power learning

First spin cost quote:

- MPW: **€12k-€25k**
- 30 packaged chips: **€1.5k-€3k**

- Sample: **€2k-€4k**
- 2× Eval-Boards & Iteration: **€2k-€4k**
- Other: **€1k**
- **Total without working time: €18k-€37k**→ plane **€25k-€45k** with buffer.

Second spin (8×8, 20-25 mm^2):

- MPW **€25k-€45k**, rest similar→ **€35k-€65k** without time; with time **€60k-€120k**.

4) Cost brakes / levers

- Use **open source shuttle** (Efabless) if OSH ok→ MPW fee not applicable.
- **Keep area small** (SRAMs are "cheap", but IO/PHY & large router FIFOs bloat).
- **Packaging standard QFN** instead of BGA/Ceramic.
- **DFT light** (boundary-scan, simple scan chains) saves debug days→ indirectly money.
- **Eval board modular** (socket/mezzanine) to shorten iterations.

5) Timeline (typical)

- **Design/Signoff:** 4-10 weeks (depending on team/experience)
- **Shuttle cutoff→ Tapeout:** fixed dates (monthly/quarterly)
- **Fab time:** ~10-16 weeks
- **Packaging+ Probing:** 2-6 weeks
- **Bring-up:** 2-6 weeks
  → **From start to executable chips: ~4-8 months**.

TL;DR "What should I budget for?"

- **Minimal (community shuttle): €0-€1.5k**
- **Solid MPW firstling (small die): €20k-€45k**
- **Ambitious proto (larger die, more tests): €40k-€70k**
- **With development time included: €60k-€120k**

🔓 Variant A: Open source (e.g. Efabless/Google Open MPW, SkyWater 130 nm)

- **Shuttle fee:** €0
  (Google/Efabless sponsor - condition: *Design completely open source*, incl. RTL, layout, possibly documentation)
- **Chip return delivery:** ~20-100 dies, often 10-40 packaged

- **Additional costs (packaging beyond freebies, bring-up, boards, measurement technology):** €500-€2 000
- **Total (without development time): €0-€2 000**
- **With development time (2-3 person months, self-made):** €10 000-€30 000
- **Risk:** You reveal your architecture to the public (anyone can view, copy and fork your RTL).

☞ **Great for hobbies, research, teaching or marketing**. **Less good** if you want to protect intellectual property.

� Option B: Closed source (commercial MPW, e.g. via Europractice, MOSIS, IMEC, TSMC, GF)

- **MPW shuttle fee (130 nm, smaller die ~10-20 mm^2):**
  €12 000 - €30 000
- **Larger die (~25 mm^2+):**
  €30 000 - €50 000
- **Packaging (20-50 packaged chips):** €1 500 - €4 000
- **Wafer probing (optional, but recommended):** €2 000 - €6 000
- **Eval boards (2-3 iterations):** €2 000 - €5 000
- **Other/lab stuff:** €500 - €1 500
- **Total (without development time): €20 000 - €65 000**
- **With development time (2-3 PM, market standard): €60 000 - €120 000**
- **Rights:** You keep your IP completely closed.

☞ **Suitable** if you really want to build a Harvard+MMU architecture as your own product or company IP.

�Comparison Open vs. Closed

Criterion	Open Source MPW (Efabless/Google)	Closed Source MPW (Europractice/MOSIS etc.)
Shuttle costs	**€0**	**€12k-€50k**
Rights/IP	All public	All private
Return delivery	20-100 dies, partly packed	20-100 dies, number of packed units selectable
Additional costs	~€500-€2k	~€5k-€15k
Development time	1-3 PM (depending on team)	1-3 PM (same)
Total without time	**€0-€2k**	**€20k-€65k**
Total with time	**€10k-€30k**	**€60k-€120k**
Availability	Quarterly shuttles	Quarterly shuttles

�Conclusion

- **If you just want to build the chip as a "proof of concept"** (show that your Harvard+MMU architecture is executable, no matter who sees the code)→ **Open Source MPW** =< €2 000 out of pocket.
- **If you need IP protection** (later commercial, investors, company)→ **Closed Source MPW= €20k-€65k** realistic. Roadmap:

64-core Harvard+MMU prototype (MPW Shuttles)

Now interactive! (Beta)

Here you can see the **project roadmap** as a timeline:

- **Phase 1 (0-9 months):** Open source MPW (4×4 mesh), costs **1-2 k€**
- **Phase 2 (9-18 months):** Closed-source MPW (4×4 mesh), costs **20-35 k€**
- **Phase 3 (18-27 months):** Closed-source MPW (8×8 mesh), costs **35-60 k€**

This way, you have functioning chips early on (open MPW), can gather know-how and then go into full expansion with protection (closed MPW).

☞ Would you like me to export the whole thing as a **PDF project plan** so that you can use it as a basis for your decision/presentation?

�Summary of security & architecture features

Harvard separation

- Strict separation of **instruction and data storage**
- Reduces attack surface (classic code injection more difficult)

MMU (Memory Management Unit)

- Virtual memory management
- Processes isolated→ Multi-user/server-capable

MPU (Memory Protection Unit)

- Protect hardware regions in the memory
- Ideal for **real-time/embedded** without full MMU overhead

NX-Bit / W^X (Write XOR Execute)

- No memory area may be **writable and executable** at the same time
- Stops shellcode injection and self-modifying code

ASLR (Address Space Layout Randomization)

- Process addresses randomized→ makes exploits more difficult

- Can be used in combination with MMU

Stack Canaries

- Protection against stack overflows
- Random value between stack frame and return address
- If value is overwritten → Program terminates safely

� Basic idea

- **MMU (Memory Management Unit):**
    - Works with **virtual addresses** → Translation virtual → physical.
    - Enables **virtual memory, paging, ASLR, process isolation**.
    - Heavy hardware overhead, often for OS with multitasking.
- **MPU (Memory Protection Unit):**
    - Works directly on **physical addresses** → no translation.
    - Defines **regions** (e.g. "0x0000-0x1FFF= read-only, 0x2000-0x2FFF= no-exec").
    - Simpler, deterministic, good for **embedded / real-time**.

�� Combination MMU+ MPU

Normally: **either or**

- **Classic server/desktop CPUs: MMU** only, because it covers everything (paging+ protection).
- **Microcontrollers (ARM Cortex-M, AVR32, etc.): MPU** only, because it is smaller, less overhead.

☞ But: **You can combine both** - and it makes sense in special scenarios:

When MPU makes sense in addition to MMU

1. **Real-time / determinism**
    - MMU can be unpredictable due to page faults or TLB misses.
    - MPU regions are deterministic (always checked at the same speed).
2. **Double protection (Defense in Depth)**
    - Even if MMU accidentally maps a page incorrectly, the MPU zone blocks.
    - Example: Kernel memory has MPU "read-only" lock, regardless of what the MMU does.
3. **Fast protection changes**
    - MPU regions can often be switched with just a few registers.
    - Interesting for threads without a full MMU context switch (e.g. lightweight RTOS tasks).

Pros & cons

�Why you don't always build separate memory areas right away

1. Flexibility

- With an MPU, you can divide **one and the same RAM chip** into different areas (RW, RO, NX).
- With "real physical separation" (e.g. separate chips), you would have to provide a separate memory block **for each category (code, data, stack, IO)**→ inflexible, more expensive.

2. Costs & space

- A shared RAM+ MPU costs less silicon & PCB area.
- Separate memories (instruction RAM, data RAM, stack RAM, IO RAM ...) would require much more chip area (SRAM blocks, buses, controllers) and external pins.

3. Compatibility

- Operating systems and compilers are traditionally designed for a **uniform address space**.
- MPU is a compromise: uniform address space, but still **hardware protection**.

4. Granularity

- An MPU allows e.g:
    - Code area RO+X
    - Heap RW, but NX
    - Stack RW, but NX
    - MMIO RW, but non-cacheable
- With hard-wired memory separations, this would be very rough and inflexible.

�When physically separated memory areas are *nevertheless* useful

- **Harvard architecture** (instructions and data physically separated)→ prevents code injection "by design".
- **Security modules / smartcards**→ ROM chip (code)+ RAM chip (data) to make attacks more difficult.
- **High-integrity real-time (avionics, automotive)**→ separate TCMs (Tightly Coupled Memories), deterministic and secure.

�Conclusion

- **MPU**= more flexible, cheaper, sufficient for embedded/mainstream CPUs.
- **Physically separated memory areas**= safer, more deterministic, but inflexible and more expensive.
- Harvard+MMU approach is actually a **middle way**: instructions and data **physically separated** (Harvard), plus **MMU/MPU rules** for more security.

Memory Operating system

📦 1. ROM (Mask ROM or OTP-ROM)

+ -Highest security: unchangeable content, ideal for bootloaders or trusted code
+ -Cheap per bit (for large quantities, because burned into the wafer)

- - No updates possible
- - Only useful if the code really never changes (e.g. minimal bootloader, crypto key routines)

�2. EEPROM

+ -Can be read like ROM, but can also be reprogrammed in the field
+ -Significantly more durable than flash (typically 10^5-10^6 write cycles)
+ -Programmable byte by byte→ Flexible for configurations

- - Small memory (kilobytes, rarely >1-2 MB on-chip)
- - Slow write speed
- - More expensive to manufacture / per bit than Flash or SRAM

☞ EEPROM is great for **configuration data, keys, settings**, but **not** as memory for a complete OS (too small+ slow).

⚡ 3. flash (NOR/NAND)

+ -Today standard for firmware/OS images
+ -MB-GB sizes possible, significantly cheaper than EEPROM
+ -Updates possible (over-the-air, bootloader with dual image)

- - Limited service life (10^4-10^5 write cycles)
- - Block-by-block erasure (less favorable than EEPROMs byte-by-byte)

☞ Typical: **Boot ROM (unchangeable)+ Flash (updateable)**

- Boot ROM contains only minimal code: loads and verifies OS from flash.
- Flash can be updated flexibly, secured by signatures.

�4. hybrid solution (best of both worlds)

- **Boot ROM (small, unchangeable)**→ guarantees that you can always boot.
- **EEPROM (small, flexible)**→ secure keys, configuration bits, security flags.
- **Flash (large, updatable)**→ Operating system and applications.

�Conclusion

- **Only ROM**= too inflexible.

- **EEPROM**= perfect for small, safety-critical data, but too small/slow for an OS.
- **Flash**= realistic choice for OS memory, plus ROM/EEPROM as a supplement.
- Combination is standard in almost all SoCs today.

Pragmatic setup (robust & small)

- **ROM (small):** Bootloader+ Signature check
- **SPI-EEPROM 4 MB:**
    - `kernel` (gzip/lzma, ~800 kB-1.6 MB)
    - `rootfs.squashfs` (BusyBox+ network+ your service, ~1-2.5 MB)
    - `cfg` (a few kB)
- **RAM:** tmpfs `overlay` for writable data
- **Optional:** small QSPI-NOR flash partition (A/B slots) for atomic updates

Why EEPROM vs. fli4l diskette anyway?

- **More reliable & more durable** than floppy disk (�), currentless data-proof.
- **Faster boot** with SPI-EEPROM (a few seconds, image→ RAM).
- **Signed updates** possible (ROM checks hash/signature).

�Advantages of CD-ROM (or "read-only media" in general)

- **Immutability:** code cannot be "secretly" changed→ very secure.
- **Costs:** CD-R costs pennies.
- **Space:** 650-700 MB→ more than enough for a minimal OS+ services.
- **Security:** "WORM" (Write Once, Read Many) - cannot be changed after burning.
- **Update concept:** New version= simply burn a new CD→ old system remains as a backup.

⚠�Disadvantages / hurdles

- **Mechanics:** CD-ROM drives are large, slow, vulnerable (vibration, power).
- **Access time:** Boot takes longer (spin-up+ read time).
- **Industry today:** optical drives almost extinct→ impractical in embedded or mini-servers.
- **Writable areas:** services need logs, config, etc.→ you need extra memory (RAM disk or flash) for this.

�Modern alternative (same principle, without drive)

- **Do not** store **ISO image on CD, but in flash / EEPROM**→ mount read-only like a "virtual CD".
- **A/B images possible:** Flash in two partitions, boot ROM loads one→ at a time corresponds to CD change.
- **RAM disk overlay:** Writable always volatile in RAM→ like fli4l back then.
- Optional: Write logs/configs regularly to small, robust flash memory (EEPROM / SPI flash with wear leveling).

☞ Conclusion:

- Today, **real CD-ROM** is more nostalgically charming than practical.
- But the **principle of "OS as a read-only image+ overlay in RAM"** is absolutely up-to-date - and can be implemented perfectly with SPI flash or EEPROM.

Architecture & goals

- **Use cases & KPIs**: Throughput target (≈ "4 GHz equivalent"), latency, power budget, area target.
- **Core topology**: Tile count, NoC topology (2D mesh), routing (XY), flow control (credits/FIFOs).
- **Harvard+ MMU**: I-/D-paths, I-TLB/D-TLB, page sizes, privilege levels, exception paths.
- **MPU?** (only if deterministic protection areas are also useful).

Memory & Boot

- **Memory plan**: ROM (Boot), CD-ROM (ISO9660), RAM sizes, optional EEPROM/USB for Config.
- **Boot chain**: ROM→ El Torito/ISOLINUX/GRUB (CD)→ Kernel→ RootFS (read-only)→ RAM overlay.
- **Recovery path**: Fallback kernel/rescue ISO, "boot next only once", clear rollback strategy.
- **Persistence**: Where are config/keys/logs located? (EEPROM/USB/Netboot/NVRAM), wear leveling if flash is used.

security

- **Secure Boot**: Root-of-Trust (ROM), signatures (ED25519/ECDSA), key rotation/revocation.
- **W^X / NX** everywhere, **ASLR**, **stack canaries**, Fortify source, RELRO/PIE in the toolchain preset.
- **Isolation**: MMU policies per process/service, least privilege, seccomp/caps (if Linux userland).
- **Entropy/TRNG**: Source for secure seeds (HW-TRNG or combined sources).
- **Update path**: Sign new ISO, rollout/backout plan (CD change process+ Audit trail).

OS & Runtime

- **SMP scheduler model**: pinned threads, run-to-completion, work stealing only controlled.
- **Message passing API**: MPSC/MPMC queues, fixed packet formats, zero-copy, backpressure.

- **FS layout**: read-only root (squashfs/iso9660), tmpfs overlay, persistent config partition separately.
- **Observability**: metrics, logs (ring buffer in RAM), remote debug (serial console/netconsole).

Verification & test

- **RTL tests**: unit tests (ALU/MMU/NoC), formal checks (arbiter/deadlock-free).
- **SoC sims**: NoC stress (hotspot/uniform), fault injection (link drops, bit flips).
- **Gate-level+ SDF** for critical paths, especially I-TLB/D-TLB ↔ caches ↔ SRAMs.
- **DFT**: Scan/BIST for SRAMs, boundary scan/JTAG plan, bring-up hooks (ROM-UART, LEDs).

Implementation (130 nm)

- **Timing closure**: STA @ slow/fast, CTS skew, critical paths (ALU bypass, branch, I-fetch).
- **Floorplan/PDN**: short paths Core↔Regfile↔SRAM, IR drop/EM, decaps, antenna fixes.
- **PVT corners**: temp/voltage/process, clock margins, reset scenarios, brown-out behavior.
- **Power/thermal**: Worst-case power, cooling concept, hot-spot analysis.

Board & mechanics

- **CD boot**: BIOS/UEFI compatibility, El Torito image tested, optical drive (SATA/USB) selection.
- **I/O**: Serial console, Ethernet (PHY/MDIO), debug header, current paths (sequencing/PGOOD).
- **EMC/ESD**: Protective circuits, filters, grounding concept, housing/vibrations (drive!).

Production & organization

- **MPW slot** (deadlines!), quantities, packaging (QFN/BGA), wafer probe plan.
- **Eval board**: Rev-A/B plan, test jigs, bring-up checklist.
- **License/IP**: Open vs. closed scope clearly defined, third-party IP (SRAM/PHY) licenses.
- **Documentation**: HW spec, memory map, boot flow, security policy, operator manual. If you

like, I can create a printable checklist for you immediately:

- a **printable checklist (PDF)**,
- **templates** for boot policy (secure boot keys/rotation),
- and a **bring-up runbook** (serial console, smoke tests, pass/fail matrix).

Quantum Random Number Generator (QRNG) as entropy source

- would immediately raise the design to "security research level".

�Why HW entropy at all?

- Everything you add in terms of security (ASLR, stack canaries, session keys, signatures) **depends on the quality of the random numbers**.
- Pseudo-RNG (PRNG) is ok, but only if it is initialized with **real entropy**.
- Many attacks of the last 20 years (Debian OpenSSL bug, Dual_EC_DRBG etc.) have exploited exactly this weak point.

⚡ QRNG vs. classic TRNG

TRNG (classic, e.g. ring oscillator jitter, noise)

+ -Easy to integrate on-chip (few gates, RC networks).
+ -Small, cheap, standard for ARM, RISC-V SoCs.

- - Bias→ must be processed by hashing/whitening.
- - Vulnerable (e.g. EM interference, clock injection).

QRNG (e.g. photon splitter, quantum noise, Zener diodes with avalanche effect)

+ -Fundamental unpredictability (quantum mechanics).
+ -Hardly manipulable if well built.
+ -Already commercially available today (e.g. ID Quantique, QuantumDice).

- - Mostly external component (optical, photodetector). On-chip hardly feasible without analog block.
- - More expensive, bulkier, requires more power.

�For your project (Harvard+MMU server chip)

- A **classic TRNG** (Jitter, Avalanche) on-chip is realistically→ small & inexpensive.
- You can connect a **QRNG** as an **external module** via USB, IZC or SPI→ that would be extremely exciting for "Backer Premium Tier" ("our board provides real quantum random numbers for your keys").
- For security-oriented backers, you can even combine both: **TRNG (on-chip)+ QRNG (optional external)→ merged and hashed (Fortuna, SHAKE-256)**.

�Conclusion

- **MVP / Prototype:** TRNG is enough, otherwise you will never get your MPW design through the synthesis flow.
- **High-end variant / Kickstarter Stretch Goal:** QRNG as an external module→ *unique selling point* for security/nerd community.

What is a USB QRNG?

An external **Quantum Random Number Generator** that supplies a bit stream via **USB** (usually as CDC-ACM / serial, HID or own driver). Advantage: real physical entropy, no housing opening required.

Integration under Linux (server/headless)

1. Plug in the device→ Kernel reports e.g. `/dev/ttyACM0` or directly `/dev/hwrng`.

2. Use **rng-tools** to push the bitstream into the kernel pool:

```
bash
sudo apt-get install rng-tools sudo
rngd -f -r /dev/hwrng
# if as serial interface: sudo rngd -f -
r /dev/ttyACM0 -W 2048
```

Tips:

- `-W 2048` (Watermark) prevents flooding of the pool.
- Permanently via systemd:

```
ini
# /etc/systemd/system/rngd-usbqrng.service [Unit]
Description=Feed kernel RNG from USB-QRNG
After=dev-hwrng.device

[Service]
ExecStart=/usr/sbin/rngd -f -r /dev/hwrng -W 2048
Restart=always

[Install]
WantedBy=multi-user.target
```

Check quality

- Activate **online tests** (`rngd` performs FIPS health checks).
- Test externally (only for demo/lab purposes, not production-blocking):

```
bash
# directly from the device into the tests (can be
slow) cat /dev/ttyACM0 | dieharder -a -g 201
```

- For NIST SP800-90B/STS: collect long captures and check offline.

Throughput/performance (default values)

- USB QRNGs for practical use: **1-10 Mbit/s**, sometimes more.
- For OS entropy (ASLR, canaries, seeds), **~100 kbit/s** is easily enough.
- Higher rates are nice for keys/CS-PRNG seeding, but not absolutely necessary.

Architecture best practices

- **Never trust a single device**: Mix sources (on-chip TRNG+ USB-QRNG+ Timing/Jitter).
  → The kernel pool is **hashed/mixed** anyway.
- Switch on **health checks** (SP800-90B-like tests, FIPS KATs/CTRs).
- **Do not** blindly trust **black-box post-processing**; documented raw bit path+ open whitening (SHA-256/SHAKE-256) is ideal.
- **Fail-safe**: If the USB QRNG fails, the boot should not hang. start `rngd` in the background, do not block in the critical boot path...

Short conclusion

- **Yes**: USB-QRNG fits perfectly with your "without unscrewing" approach.

- Integration is **simple**: `rngd`+ udev/systemd.
- Best security: **QRNG+ mix on-chip TRNG**, health checks active, no single point of failure.

If you like, I'll build you the udev **rule** and a **ready-made systemd unit** that automatically recognizes /dev/ttyACM* `QRNGs` and starts `rngd` cleanly.

You:
context limit reached, right? we are still talking about a dedicated server-os, as linux would not work with our architecture.

ChatGPT:
good catch - yes, we're strictly sticking with **our own server OS** (because your Harvard+MMU architecture is not Linux compatible). So: how do you integrate **USB-QRNG** cleanly **into your own OS**?

Target image (short)

- **USB host stack** (Control+ Bulk/HID)→ **QRNG driver**→ **Entropy daemon in kernel**→ **CSPRNG**→ OS-Wide: `getrandom()`//dev/random-equivalent.
- **Mixing of several sources** (on-chip TRNG, QRNG, timer jitter if necessary).
- **Health checks**+ **Whitening**+ **Conservative entropy evaluation**.
- **Fail-safe**: Boot does not block, but safety-critical services wait for "enough entropy".

---

1) USB device model & protocol

Two simple paths that you control yourself:

A. CDC-ACM (virtual serial)

- QRNG sends frames via bulk-IN.
- Framing e.g.: `[MAGIC=0x51 0x52], LEN(2B), PAYLOAD, CRC32(4B)`
- Advantage: Minimal driver (read ACM only).
- Disadvantage: No standardized hwrng `class code` - but does not matter in your own OS.

B. Vendor-specific bulk

- Own interface class, only bulk-IN EP.
- Simple data transfer, no TTY overhead.

Crucial: **unique product IDs**+ **small descriptor feature** (e.g. iProduct "QRNG-v1") so that your OS recognizes it automatically.

---

2) Kernel modules (layers)

usb_core

- Enumeration, endpoint open, URB/transfer queue, error paths.

qrng_drv

- Recognizes device (VID/PID/strings).
- Reads **frames** via bulk-IN; performs **frame checks** (LEN, CRC).

- Delivers **raw bits** to `entropy_mgr`.

entropy_mgr

- **Health checks (online)**: Repetition Count, Adaptive Proportion (SP800-90B style), simple monobit/runs checks.
- **Whitening**: Hash-based compression (SHA-256/Keccak-SHAKE256).
- **Entropy estimate** per incoming block conservative (e.g. 0.5-0.9 bit/bit depending on tests).
- **Pool**: ring buffer (e.g. 64-256 KiB)+ **credit counter** (estimated entropy bits).

csprng

- Implements DRBG (recommended: **ChaCha20-DRBG** or **AES-CTR-DRBG**).
- **Seeding**: min. 256 bits of verified entropy.
- **Reseed**: cyclic (e.g. data set/timer) or on-demand, merges new hash from pool.
- API: `os_getrandom(void* buf, size_t n, FLAGS)`; Blocked if `FLAGS&BLOCK` and entropy< minimum level.

api/devfs

- `/dev/random` (blocking, "real" entropy)
- `/dev/urandom` (CSPRNG output, non-blocking)
- Syscall `getrandom()`-equivalent.

---

3) Reseed policy (proven)

- **Initial**: wait for ≥**256 bits of** valued entropy.
- **Periodic**: every **64 MiB** of output data **or** every **60 s**, whichever comes first.
- **Event-driven**: When new QRNG block with high credit comes in.
- **Mixture**: `new_seed= H(old_key|| pool_extract|| time_nonce)`

---

4) Health checks (lightweight)

- **Repetition Count Test**: abort if too many identical bits in a row.
- **Adaptive proportion** (window N=1024): Proportion '1' must not be <0.36 or >0.64, for example.
- **Frame integrity**: CRC32+ incremental counters; drop duplicate/missing sequences.
- **Fail-degrade**: If QRNG log bad→ , **do not** "poison" the CSPRNG. Continue with other sources.

---

5) Multi-source mixture

- **on-chip TRNG** (Jitter/Avalanche)→ low but steady entropy.

- **USB-QRNG** → Burst-wise high entropy.
- **Timing jitter** (interrupt jitter/network timestamps) → small additive contributions.
- **Mixing:** always via Hash-Compress (SHA-256/SHAKE256), **no** linear XORs alone.

6) Boot flow (CD-ROM OS)

1. **Boot ROM** → **Kernel** (from CD).
2. `entropy_mgr` starts early, collects on-chip TRNG.
3. **USB stack up** → `qrng_drv` tries to claim QRNG.
4. **CSPRNG seeds** as soon as `≥256` bits verified.
5. **Security-critical services** (Keygen/TLS) are waiting for `SEED_READY`.
6. System running; **reseed** according to policy.

7) SMP aspects

- `entropy_mgr` and `csprng` should be **lock-poor**:
  - Extraction ring: **MPSC/MPMC ring** with per-CPU reserves.
  - `getrandom()` could have per-CPU DRBG instances (all are periodically **reseeded** with the global pool) to minimize contention.

8) Minimal spec & pseudocode

QRNG frame (from device):

```
rust
struct qrng_frame {
  u8 magic[2];     // 0x51 0x52
  u16 len;         // payload bytes (e.g. 512)
  u32 seq;         // monotonically increasing
  sequence number u8 payload[len];
  u32 crc32;
}
```

Kernel driver (pseudo-C):

```
c
void qrng_read_loop(ep_bulk_in) {
  while (dev_present) {
    int n= usb_bulk_read(ep_bulk_in, buf, sizeof(buf), TIMEOUT); if
    (n <= MIN_HDR) continue;
    if (!check_magic_len_crc(buf,n)) continue; struct
    qrng_frame *f = (void*)buf; entropy_mgr_submit(f-
    >payload, f->len, f->seq);
  }
}

void entropy_mgr_submit(const u8* p, size_t n, u32 seq) { if
  (!health_ok(p,n)) { stats.bad++; return; }
  // Whitening: hash to 32..64B digest to avoid bias/correlation u8
  digest[32];
  sha256(p,n,digest);
  ring_push(&pool, digest, sizeof(digest));
  credits+= estimate_entropy_bits(digest, sizeof(digest)); // conservative if
  (need_reseed() && credits >= 256) csprng_reseed(extract_pool());
}

ssize_t os_getrandom(void* out, size_t n, int flags) {
  if ((flags&BLOCK) && !csprng_seeded()) wait_event(seed_ready);
  csprng_generate(out, n);
  return n;
}
```

---

9) Tests & validation

- **Offline**: long captures of the QRNG by **SP800-90B/STS** and **dieharder**.
- **Online**: activates repetition/adaptive proportion; log+ counters.
- **Fault injection**: throttled QRNG, frozen bits, replay frames → OS must react robustly.
- **Throughput target**: 1-10 Mbit/s is sufficient; select buffer so that no USB dropouts occur (e.g. 64 KiB double buffered).

---

10) Practical tips

- Optional AES-GCM-packed frames **on the device side** → Tamper protection on the bus (key from device ROM).
- **Power/EMI robustness**: USB hub with good power supply; QRNG often sensitive.
- **Config flag**: "Require QRNG for Seed?" → Allow boot-blocking in safety-critical environments, otherwise not.

Phase 0 - Architecture & Targets (2-3 weeks)

- **Targets/KPIs:** 4 GHz-equivalent *throughput line*, clock/core, cores× IPC, power budget.
- **Specification v1:** Harvard+ MMU, 2D mesh, scratchpads, CD-ROM bootflow, security preset (NX/W^X, ASLR, Canaries).
- **Deliverables:** 10-page Arch-Spec, Memory-Map, Boot-Flow (CD-ROM), Interrupt-/Trap-Matrix.
- **Exit:** all interfaces frozen (tile↔NoC, MMU API, OS syscalls).

Phase 1 - Simulator & OS core (4-6 weeks)

- **Core-ISS/Emu:** functional model (C++/Python) with I/D-Harvard, TLB/TLB-Miss, exceptions.
- **Own server OS (MVP):** process/thread, scheduler (pinned), IPC (message queues), VFS (iso9660 ro+ tmpfs overlay), RNG-API.
- **Tooling:** Assembler/Linker minimal, C-Runtime, Build-System.
- **Exit:** Boot from **ISO** in emu to "hello service", IPC ping, `getrandom()` returns bits (pseudo).

Phase 2 - RTL core & NoC (6-8 weeks)

- **RTL:** 5-stage pipeline, MMU (SV39-like or fixed), 4 KiB-SPM/tile, DMA stub, router (RR+ FIFOs).
- **Timing target:** 80-100 MHz @ 130 nm (OpenROAD flow); STA on slow/fast corners.
- **Verification:** Unit TBs (ALU/MMU/Router), NoC stress (hotspot/uniform), simple formal checks (deadlock-free).
- **Exit:** RTL "feature-complete", synth OK, top 10 critical paths documented.

Phase 3 - FPGA bring-up (4-6 weeks)

- **Porting:** 4×4 mesh to FPGA (2-3 boards, e.g. Artix-7/Kintex-7).
- **OS port:** CD-ROM image as initrd/ROM-FS, driver for UART/Eth-MAC (if available).
- **Measurements:** Throughput/latency, scheduler overhead, NoC saturation, IRQ jitter.
- **Exit:** End-to-end: Boot→ Services→ Parallel workload reaches target throughput (phase-related).

Phase 4 - MPW Proto A (4×4, Open or Closed) (6-9 months incl. Fab)

- **Tapeout preparation:** DRC/LVS clean, antenna fixes, scan/BIST light, pads/ESD, power grid.
- **Security anchor:** Boot ROM (small), signature check for ISO; NX/W^X strict; ASLR/Canaries in toolchain preset.
- **Board Rev-A:** Serial console, 1× GbE, SATA/USB-CD-ROM (or USB-CD adapter), debug header.
- **Exit:** Silicon bring-up: CD boot, scheduler/IPC tests, simple services stable.

Phase 5 - MPW Proto B (8×8, Closed) (6-9 months)

- **Scale-up:** Router buffer, flow control (credits), DMA full; per-CPU DRBG, entropy pool (optional USB-QRNG).
- **Performance:** Target "~4 GHz-equiv. throughput" (e.g. 64×80 MHz×~1 IPC) on parallelizable jobs.
- **Reliability:** Brown-out/reset matrix, watchdog path, error telemetry.
- **Exit:** Benchmarks (VecAdd/FFT/MapReduce-like)≥ Target, endurance tests 72 h green.

Cross tasks (run in parallel)

- **Security:** Secure boot keys/rotation, signature tool, policy "require entropy for keygen", audit logs in RAM.
- **Updates:** CD image versioning, rollback process ("next boot once").
- **Docs:** HW-Spec, OS-Syscalls, Driver-ABIs, Bring-Up-Runbook.

Resources & costs (thumbs)

- **Team:** 2-3 Eng. (RTL/PD), 1 Eng. (OS/Runtime), 0.5 PM/QA.
- **MPW (4×4):** 20-35 k€ closed (0-2 k€ open). **MPW (8×8):** 35-60 k€ closed.
- **Boards/tools:** 3-8 k€.

Risks & Mitigation

- **Timing @130 nm**: decouple critical paths early (Mul/Div multicyclic, Branch early), SPM synchronous, CTS conservative.
- **NoC deadlocks**: formal properties+ Livelock tests; credits instead of blind ready/valid.
- **Entropy failure**: multi-source (on-chip TRNG+ optional USB-QRNG), non-blocking boot.
- **CD-ROM boot**: test El-Torito/BIOS & UEFI, alternate path via USB-CD adapter.

---

1. What is RVA23?

- "RVA" is the **RISC-V architecture versioning abbreviation**.
- **RVA23**= "RISC-V Profiles, 2023 generation", i.e. the **official minimum feature set** that RISC-V servers/embedded chips should comply with from 2023.
- Example: **RVA23U64** (user profile for 64-bit CPUs) or **RVA23S64** (supervisor profile).
- Contains standardized mandatory extensions (M= Mul/Div, A= Atomics, C= Compressed, Zicsr, Zifencei etc.) and security features (MMU, PMP, optional hypervisor).

---

2. Security compared to your Harvard+MMU design

Your custom ISA approach had the following key points:

- **Harvard separation** (Instr./Data physically separated).
- **MMU** with PTE flags R/W/X/U/G→ NX enforced, W^X.
- **ASLR & Canaries** through OS.
- **Optional MPU-style regions** or ROM kernel.

RVA23 (RISC-V 2023 profile):

- Based on classic **Von Neumann architecture** (Instr.+ data in the same RAM, caches unified).
- But has **equivalent security primitives**:
  - **MMU** with per-page **R/W/X/U** flags→ NX and W^X can be enforced.
  - **PMP/Smepmp** (Physical Memory Protection)→ works similar to an MPU, even without MMU active.
  - **ASLR**: possible through OS page shifts.
  - **Stack Canaries**: OS/Compiler feature, like yours.
- Advantage: Everything already standardized→ Toolchain, compiler, OS support (Linux, BSD, seL4, etc.) available.

---

3. Differences

- **Harvard vs. Von Neumann**:
  Harvard provides somewhat more robustness against classic "data-as-code" (code injection). With RISC-V you need the NX bit→ same effect, only at MMU level.
- **Complexity**:
  Your ISA would be very small (easier to audit). RVA23 is larger, but also already battle-tested.

- **Ecosystem**:
  You can compile RVA23 immediately with LLVM/GCC, with your ISA you would have to build toolchains yourself.

---

4. Conclusion

- **Yes, RVA23 is a roughly equally safe alternative** if you:
  - configure the OS so that **W^X, NX, ASLR** are active,
  - optionally use PMP/MPU for additional isolation.
- Your **Harvard design** would be more "secure by construction", while RVA23 is "secure by policy/flags".
- But RISC-V is **practically usable** (existing toolchain, Linux/BSD support).

☞ In short:

- If you *really* want something custom, minimal, audited → your Harvard design.
- If you want "realistic, with ecosystem and toolchains" → RVA23.

Shall I make you a **table overview of Harvard vs. RVA23 (security features, effort, toolchain, risk)** so that you can see the trade-offs at a glance?

You:
The design risks with Harvard-with-MMU are unfortunately a problem.

ChatGPT:
Totally get it - **Harvard+MMU** is technically exciting, but the **risk profile** (MMU correctness, I/D coherence, toolchain, verification) is really hefty for a solo/small team.

Pragmatist pivot (safe & feasible)

Option 1: RISC-V (RVA23) as the basis

- **Security parity**: NX/W^X, ASLR, Canaries, PMP (≈ MPU), MMU → "equally secure" with the right policy.
- **Ecosystem**: GCC/LLVM, debugger, existing OS components.
- **Your USP remains**: **immutable CD-ROM OS**, **server minimalism**, **USB QRNG** (optional), **SMP throughput**.

Stack suggestion

- **ISA**: RVA23S64 (or RV32 for smaller FPGAs).
- **Kernel**: small microkernel/unikernel style (own runtime) instead of Linux monolith.
- **Security preset**: W^X strict, PIE/RELRO, Canaries, ASLR, seccomp `equivalent` (Capabilities).
- **Entropy**: CSPRNG+ optional USB-QRNG; fail-safe seeding.
- **Boot**: CD-ISO (read-only)+ tmpfs overlay; Config on small external RW medium.

Option 2: Harvard without MMU (MPU/PMP-only)

- For **RT/determinism** ok, but **server isolation** weaker.
- More suitable if you have strict single-purpose services.
- I would **not** recommend it for server target.

Concrete roadmap (lean)

1. **Decision**: RVA23 (RV64/32).
2. **Hello-SMP** on FPGA/Emu with existing core (e.g. Rocket/BOOM for RV64 or PicoRV/Serv for RV32).
3. **Own server mini-OS**:
    - Processes/threads (pinned), message queues, timers, net stack (UDP/TCP mini).
    - ISO9660 boot, W^X/NX, ASLR, Canaries from day 1.
4. **Entropy subsystem**: CSPRNG+ (optional) USB QRNG driver.
5. **Immutable Deploy**: CD image, A/B slots (two ISOs), signatures.
6. **SMP throughput demo**: Map/Reduce-type jobs→ "4 GHz-equiv. throughput".

Where the big risks shrink

- **MMU correctness**: already clearly standardized in RISC-V, reference cores available.
- **Toolchain**: ready to use (no own ISA toolchain).
- **Verification**: you test your **OS/policy**, not **the ISA basics**.

If you want something more specialized later

- You can enforce **Harvard-like discipline** (strict W^X, executables only from separate segment, XIP only read-only) without customizing the hardware. If you

like, I'll put together a **pivot starter** for you now:

- minimal **RISC-V-Emu/FPGA-Hello**,
- **CD-ROM boot ISO** for your OS,
- security preset (build flags & linker script),
- and a **checklist** of what you enable for RVA23 to practically achieve the Harvard security goals.

You:

achso, how many bits does our processor actually have?

ChatGPT:

We had previously designed your design **for 32 bits**:

- **Register width:** 16 GPR à 32 bits (`r0..r15`).
- **Instruction width:** fixed 32 bits.
- **Address space:** with 4 KiB pages and 20-bit PFN→ 32-bit VA/PA (max. 4 GiB).
- **Syscall ABI:** Return/Args in 32-bit registers.

☞ So: **our Harvard+MMU processor is 32-bit**.

If you are now thinking of **RVA23 (RISC-V profiles)** rather than Pivot:

- **RV32**→ close to your design, smaller, easier to pack into the FPGA.
- **RV64**→ more future-proof (server, modern toolchains).

Do you want me to break down the **pros and cons of 32 vs. 64 bit** specifically for your "mini server OS on CD-ROM"?

You:

3GB of RAM is actually enough for most server services, right?

ChatGPT:

Absolutely👍 - for most "classic" server services, **3GB of RAM** is more than enough if you deliberately build the system minimalistically.

1. Typical services & RAM budget

- **Web server (nginx/lighttpd)**:< 50 MB for binary+ caches.
- **Database (SQLite/Postgres light)**: Hundreds of MB possible, but 1-2 GB are sufficient for small instances.
- **Mail server (Postfix/Dovecot)**: usually< 200 MB per process, peak perhaps 500 MB under high load.
- **DNS (Unbound/Bind)**: < 100 MB.
- **SSH / VPN / Proxy**: double-digit MB range.

→ With **3 GB** you can run several of these services in parallel without any problems.

2. Why more is often not needed

- **Many services are I/O-bound**, not RAM-bound.
- CD-ROM-OS+ read-only RootFS → very little footprint in memory.
- Your approach (immutable OS, no GUI, no desktop overhead) saves a massive amount of RAM.

3. When things get tight

- **Database+ caches** (Redis/Memcached) → which eat up RAM if you want to use them as an in-memory store.
- **Java/Node.js apps** → VMs/interpreters often need 1-2 GB on their own.
- **Containerization (Docker/K8s)** → layer + overhead multiply the demand.
- **Modern web stacks (Elasticsearch, Kafka, etc.)** → beyond your 3 GB mark.

4. For your use case

- Minimalistic server OS on **3 GB RAM** is **absolutely feasible** for:
  - Web+ Mail+ DNS+ small DB at the same time.
  - Dedicated special services (proxy, firewall, VPN, router).
- Your concept is **optimized for leanness and robustness** anyway, not for monster microservices.