


# risc-v

---

 A VHDL implementation of RISC-V 

Developed by David Heim as part of the summer 2022 **Prozessorbau** course at **Uni Augsburg**

## Setup

Das Vivado-Projekt wird automatisch aus TCL-Skripten generiert. Zum generieren und öffnen des Projekts:

```
$ vivado -mode tcl -source scripts/project.tcl
```

Achtung: Ein vorhandenes Projekt wird dabei überschrieben. Für ein bereits generiertes Projekt kann man in Vivado die Datei `scripts/project/risc.xpr` öffnen.

Zusätzlich zum Projekt werden auch die verwendeten IP-Module per Skript generiert. Bei Ausführung der `project.tcl` wird automatisch auch das `ip.tcl` Skript ausgeführt, welches die Konfiguration der IP-Module beschreibt. Bei Änderungen an einer Konfiguration muss der Ordner des Moduls unter `scripts/ip/[name]` gelöscht werden, damit das Modul neu generiert wird.

## Dokumentation

### Multithreading

Multithreading ist eine Form von Thread-level Parallelismus bei der sich mehrere Threads die Ausführungseinheiten eines Prozessors teilen. Jeder Thread benötigt seinen eigenen State, d.h. einen eigenen Program Counter und Register Set.

Es gibt zwei verschiedene Ansätze für Multithreading bei einer Single-Issue CPU:

- Fine-grained Multithreading

Wechselt mit jeder Instruktion zwischen den Threads, meist im Round Robin Verfahren. Dadurch können bei Stalls eines Threads Instruktionen der anderen Threads ausgeführt werden und der Prozessordurchsatz aufrecht erhalten werden. Umgekehrt laufen einzelne Threads, welche ohne Stalls ausgeführt werden, langsamer, da sie sich ihre Ausführungszeit mit den anderen Threads teilen.

- Course-grained Multithreading

Wechselt zwischen den Threads nur bei längeren, kostspieligen Stalls. Dadurch wird der Prozessor bei Stall-freien Ausführungen weniger verlangsamt, allerdings wird bei jedem Thread-Wechsel ein Pipeline-Bubble benötigt.

Dieser Prozessor implementiert Fine-grained Multithreading mit Round Robin.

### Konfiguration

Es gibt mehrere Konfigurationsmöglichkeiten, die als Generic-Parameter an der **Processor** Entität gesetzt werden:

- **ThreadCount**: Die Anzahl der Threads. Frei wählbar.
- **ThreadStart**:
  - **start\_0**: Alle Threads werden sofort gestartet und führen die selben Instruktionen, beginnend bei Adresse 0, aus.
  - **start\_offset**: Alle Threads werden sofort gestartet, führen jedoch unterschiedliche Instruktionen aus, beginnend bei einem festen Offset (z.B. Thread 0 bei 0x0, Thread 1 bei 0x100)
  - **spawn**: Nur Thread 0 wird gestartet. Weitere Threads können mit einer speziellen Instruktion an einer bestimmten Adresse starten. (s. [Zusätzliche Instruktionen](#))

## Anpassungen an der Pipeline

- **ThreadTag**  
Jede Pipeline-Stufe muss wissen, welcher Thread gerade in ihr ausgeführt wird. Ein Signal **ThreadTag** wird dafür über alle Stufen durchgereicht.
- **Register**  
Jeder Thread benötigt seinen eigenen kompletten Registersatz. Die Register werden hierfür in einem zweidimensionalen Array gespeichert, welches über ThreadTag und Register indexiert wird. Zum Lesen wird der aktuelle ThreadTag der Decode-Stufe und zum Schreiben der, der MEM-Stufe mitgegeben.
- **Clear/Interlock/Forward**  
Die jeweilige Aktion ist nur valide, wenn sie den selben Thread betrifft, der sie ausgelöst hat. Z.B. darf Thread X nur Daten geforwardet bekommen, welche von ihm selbst geschrieben wurden, auch wenn ein anderer Thread zur gleichen Zeit das gleiche Register schreibt. Auch hierfür werden die ThreadTags der jeweiligen Stufen zusätzlich zu dem eigentlichen Signal mitgegeben.
- **Fetch-Stufe**  
Damit für jeden Thread ein Program Counter gespeichert werden kann hat wurde das **PC** Signal als Array implementiert, indexierbar mit dem ThreadTag. Jeden Takt speichert die Fetch-Stufe zuerst den nächsten PC (PC+4) für den aktuellen Thread in das Array und wählt dann den nächsten (aktiven) Thread zum Ausführen und liest dessen PC aus dem Array.  
Besondere Behandlung benötigen in der Fetch-Stufe Jumps. Auch hier muss wieder der ThreadTag des den Sprung ausführenden Threads mitgegeben werden, damit dessen PC entsprechend angepasst werden kann.
- **Stalls bei Speicherzugriffen**  
Da die Memory-Einheit nicht gepipelined ist muss der Prozessor entgegen der eigentlichen Idee von Multithreading bei Speicherzugriffen weiterhin komplett gestalled werden. Dies ist der Fall, da das Thread-Wechseln in der Fetch-Stufe stattfindet. Eine mögliche Erweiterung wäre es, andere Threads, die keine Speicherzugriffe ausführen, nicht zu stallen.

## Zusätzliche Instruktionen

Es wurden zwei zusätzliche CSR-Befehle implementiert:

- ThreadTag/Thread-ID des aktuellen Threads in Register speichern:  
Lese CSR 71D ("TID", "Thread-ID"):

```
csrr 0x71D, #reg
```

- Spawnen des Threads mit `id` an der Adresse in Register `#reg`:  
Schreibe CSR 700+`id` (z.B. 702 für `id`=2)

```
la    #reg, #address  
csrw 0x702, #reg
```

Dieser Befehl ist ähnlich wie ein Jump implementiert. Die EX-Stufe reicht der Fetch-Stufe den ThreadTag des zu spawnenden Threads, sowie die Spawn-Adresse weiter. Die Fetch-Stufe schreibt dann in das PC-Array.