AXI4-Lite

Różnice względem AXI:

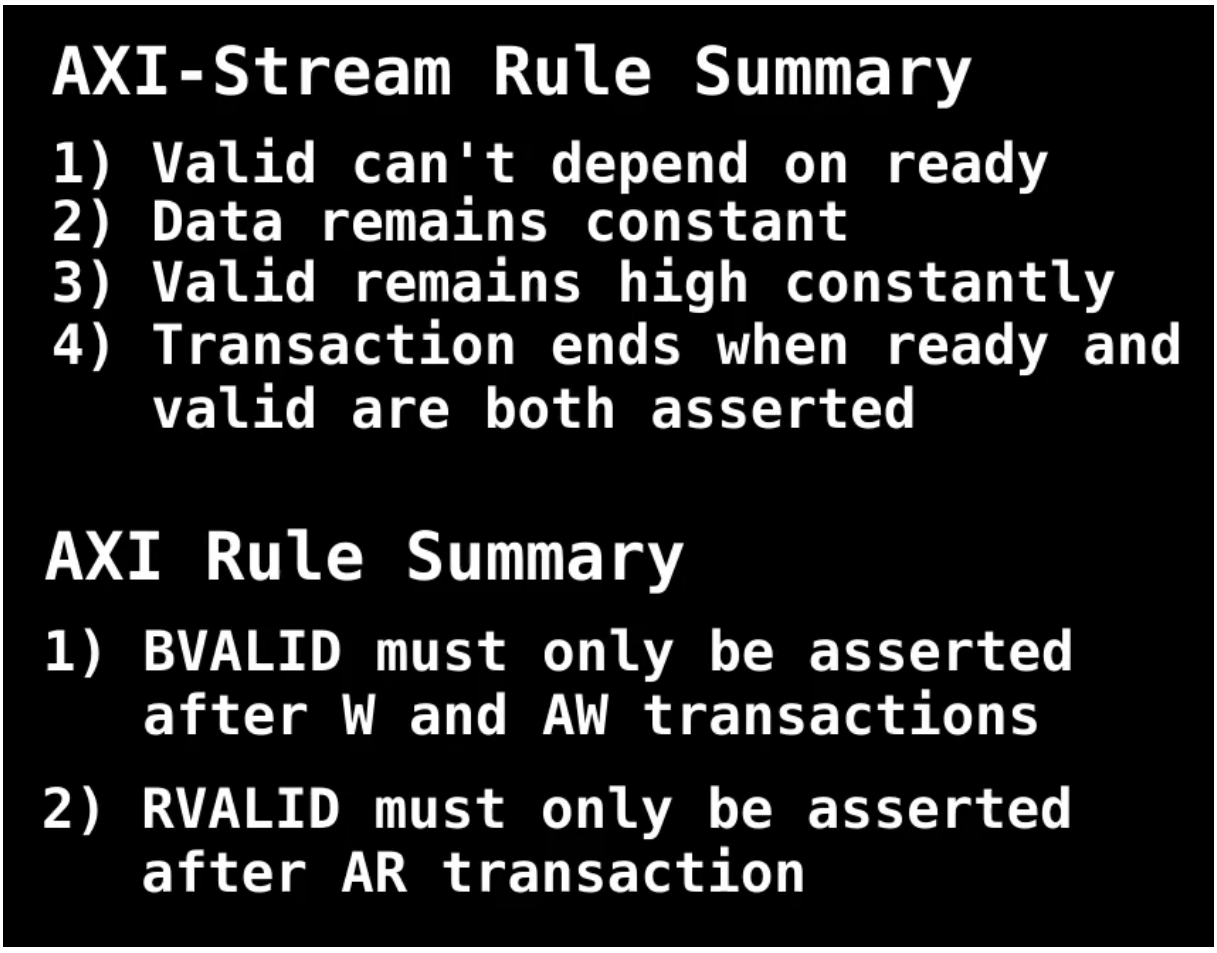
- Nie ma sygnału last
- Wielkość burst zawsze wynosi 1 – więc last nie jest potrzebny

Strobe to taka maska dla data, która mówi, które bajty mają być zapisane np. wdata = 0xFFFFFFFF i wstrb = 0b1010 to zostanie zapisane 0xFF00FF00

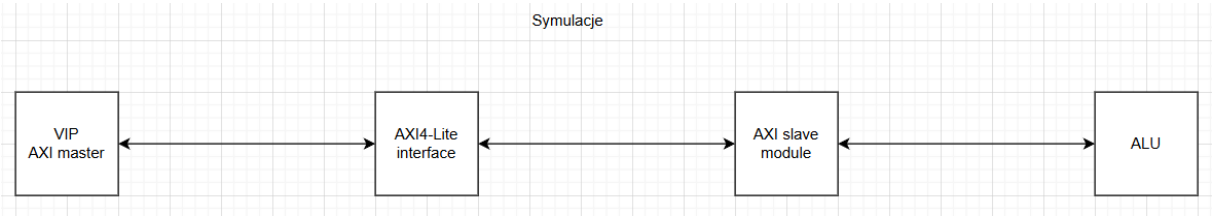
Table B1-1 AXI4-Lite interface signals

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
–	AWADDR	WDATA	BRESP	ARADDR	RDATA
–	AWPROT	WSTRB	–	ARPROT	RRESP

Zasady, których trzeba się trzymać



Okej, a więc na pierwszy etap projektu:



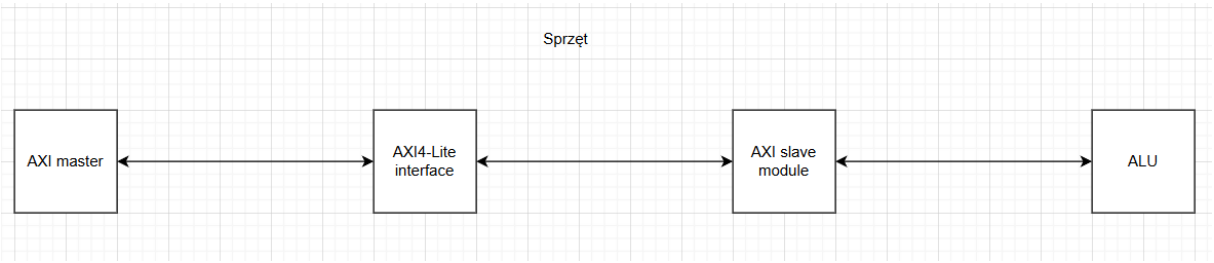
VIP master to gotowy moduł tylko do zaimplementowania, który formuje gotowe transakcje AXI i je wysyła. AXI4-Lite to główna część projektu czyli moja implementacja AXI4-Lite. AXI slave to moduł do napisania, który działa jak slave i jest dostosowany do interfejsu axi. Ma wewnątrz swoje rejestry do przechowywania danych – czyli operandów, operacji i wyniku.

ALU – prosty moduł, który na wejściu ma operandy, operacje, implementuje zadaną logikę i na wyjściu daje wynik. Jego porty są zmapowane do rejestrów axi slave'a.

Na pierwszym etapie projektu w zasadzie chcemy przetestować naszą implementację protokołu axi i slave'a. VIP będzie generował zadane transakcje, a ja sprawdzę jak zachowuje się protokół i slave. ALU będzie tylko wpisywal jakies rzeczy do resultsów.

W najwęższym przypadku trzeba sprawdzić że VIP master robi write(operandA), write(operandB), write(operacja) i read(wynik) i że to działa zgodnie z założeniem. Oczywiście trzeba potestować to dużo szerzej.

Etap drugi



Zakładamy, że interfejs, slave i ALU działają poprawnie.

Teraz potrzeba prawdziwego mastera, który robi na przykład sekwencję:

Write(opA, 3), write(opB, 7), write(op, add), read(result) i ma dostać 10. Taka sekwencja to jest implementacja SW – czyli patrzac z wysokiego poziomu abstrakcji (w ogólności np. jest to program w C – oczywiście najlepiej bare-metal, żeby nie było zadnego linuxa itp.). Żeby te instrukcje były przekazane na sprzęcie trzeba je przetłumaczyć na protokół AXI4-Lite. Do tego można wykorzystać gotowe drivery. Czyli one wyślą/ odbiorą dane dopasowane do interfejsu AXI, resztą zajmie się już moja logika – która rzekoma będzie działać już poprawnie.

Na pierwszy etap projektu trzeba napisać zatem moduły:

- ALU – mały i prosty
- AXI slave – bardziej złożony
- AXI4-Lite – największy, nie wiem czy to jeden moduł – zmartwienie na potem
- Instancja VIPa

Dobry opis

https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081

Dodać prosty bank rejestrów!!!!

Sprzęt CMOD7

Each AXI interface has a single clock signal, **ACLK**. All input signals are sampled on the rising edge of **ACLK**. All output signal changes can only occur after the rising edge of **ACLK**.

Inputy mogą być przypisane chyba niesynchronicznie, ale muszą być samplowane synchronicznie, outputy muszą być zmieniane synchronicznie

Postępy prac

Wykorzystano język system verilog – rozszerzona wersja verilog pozwalająca na wygodniejsze pisanie kodu źródłowego

Zaczęto od interfejsu axi. W tym celu stworzono prosty plik konfiguracyjny params.vh. Porty axi ‘zamknięto’ w *interface*. Stworzono dwie perspektywy – dla slave’a i mastera.

Zaczęto pisać kod modułu axi slave. Zgodnie z dokumentacja (wersja H.c) (paragraf A3.1.2) reset ma zerować na slave tylko Rvalid i Bvalid – reszta nieistotna. U nas zerujemy wszystkie sygnały – dla przejrzystości na wave.

AXI slavea zrealizowano jako maszynę stanów.

Kierowano się tym wykresem z dokumentacji (podwójna strzałka oznacza, że sygnał na jej końcu może wystąpić dopiero po sygnale na jej początku, pojedyncza oznacza, że nie musi tak być):

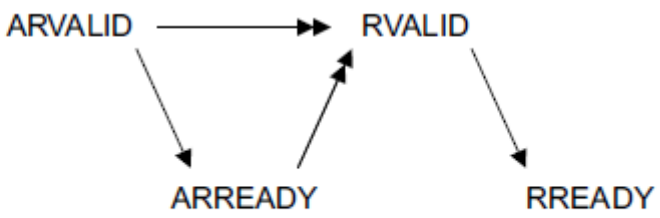


Figure A3-5 Read transaction handshake dependencies

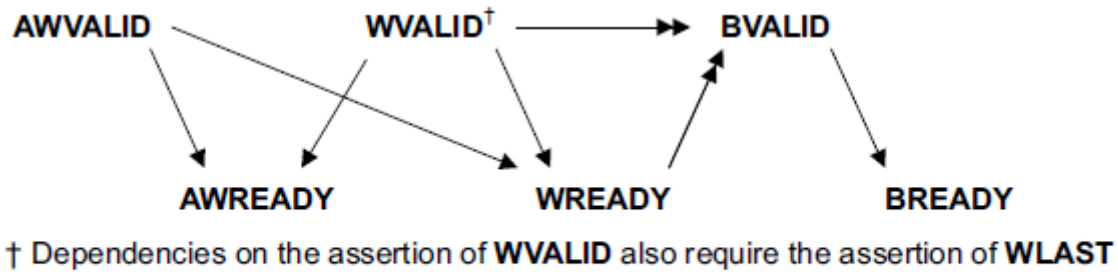


Figure A3-6 AXI3 write transaction handshake dependencies

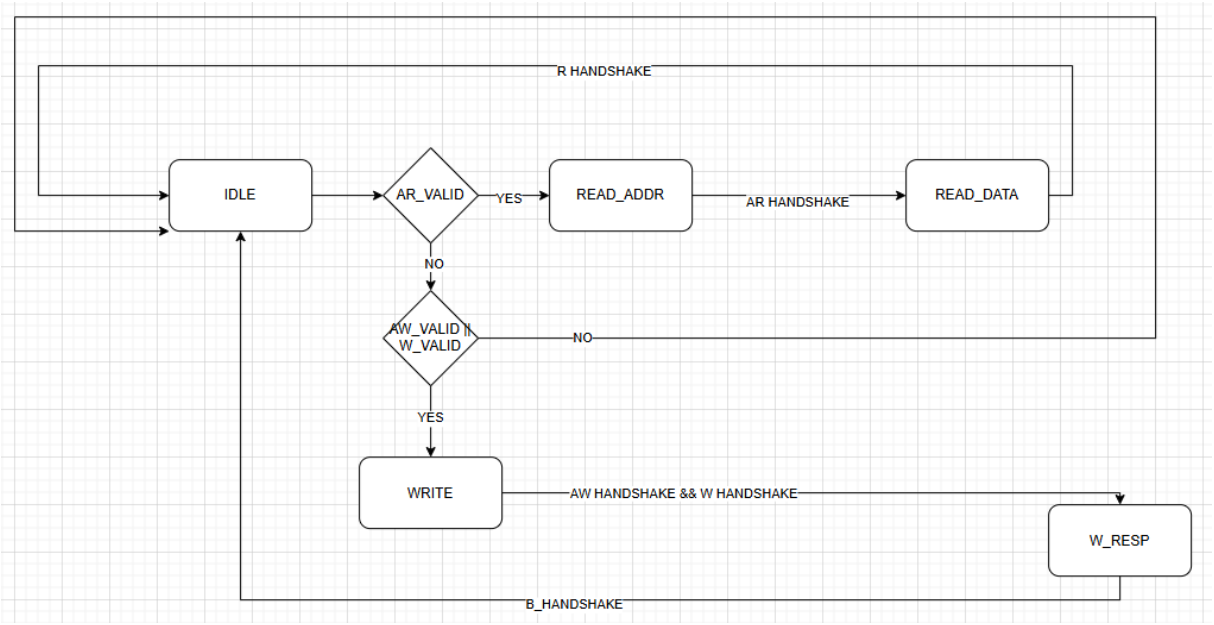
Oznacza to, że dla readów kanał R musi czekać na handshake na kanale AR.

Dla writeów kanał B musi czekać na handshake na AW **i** W (czyli na dwa handshake). AW i W mogą działać równolegle, co jest istotne.

Zgodnie z dokumentacją ready powinno być domyślnie w stanie high. – na razie zaimplementowane ze domyślnie low, tak było latwiej zakodowac

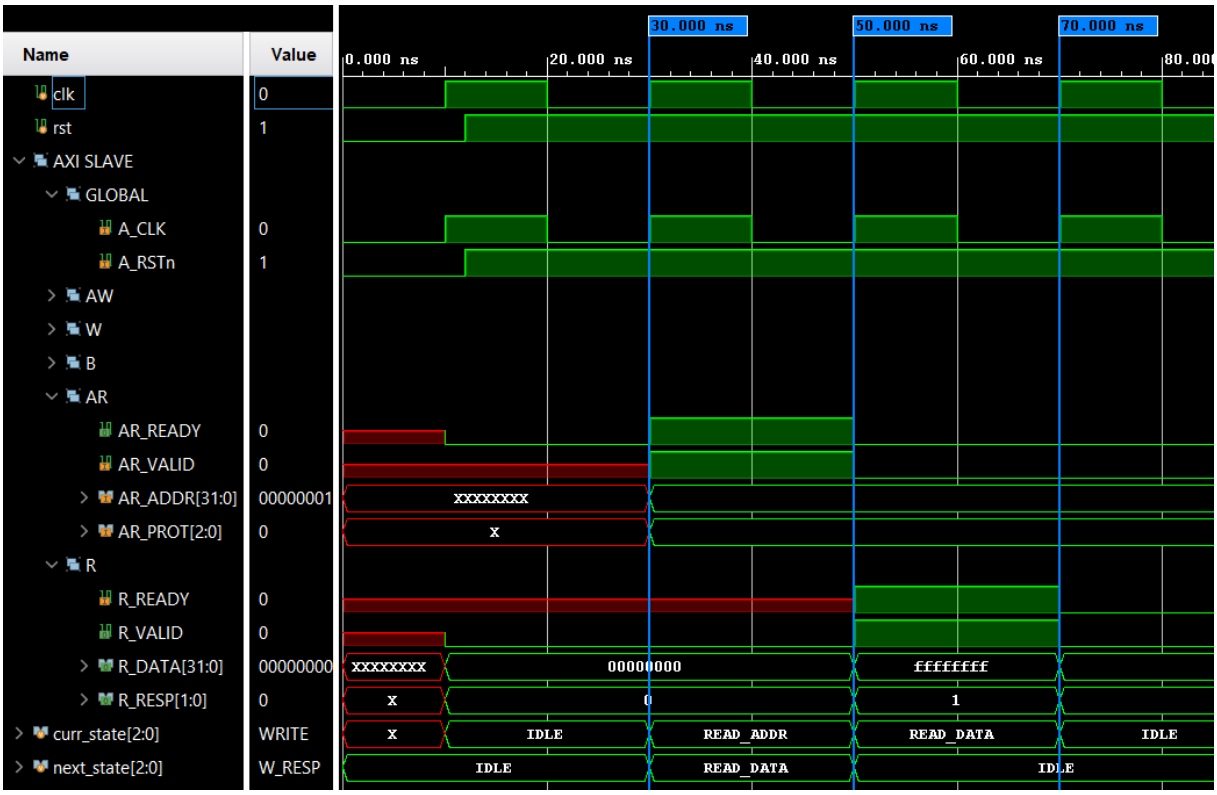
Zgodnie z dokumentacją jeśli master wystawi valid to musi czekac do końca transakcji, żeby go wyłączyć. Więc jeśli w maszynie stanów jestem w czytaniu to nie muszę się przejmować pisaniem (zrobię write po skończeniu read) i na odwrót. Dokumentacja zakłada, że powinno się priorytetyzować read (muszę znaleźć potwierdzenie na to, bo tylko tak przeczytałem w internecie).

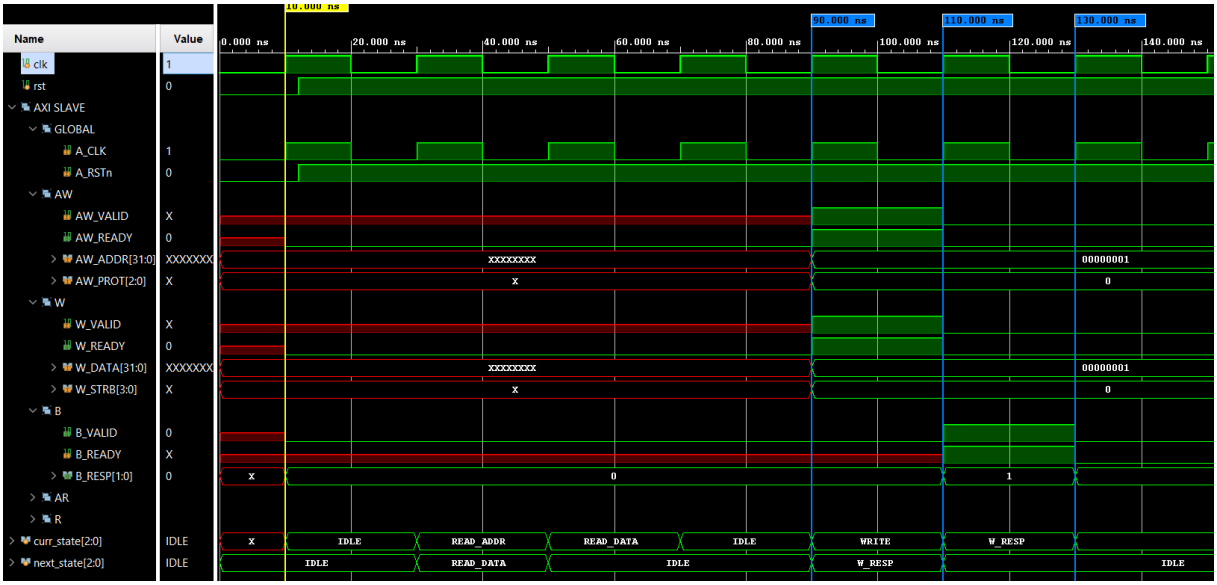
Ponieważ kolejność W, AW nie jest istotna to są obserwowane wspólnie w jednym stanie. Schemat maszyny stanów:



(Schemat do upiększenia)

Obecny stan (10.12.25) w slave nie uwzglednia zadnych rejestrow





Prosta symulacja pokazuje ze działa zgodnie z zalozeniem. Poki co dane wpisuje na sztywno – bo nie ma rejestrow (axi4lite_simple_read_test) testuje wersje bez rejestrów żadnych

Link do mojego githuba z kodem: https://github.com/codeQuanto/AXI4-Lite

Dobra dalej zaprojektowano bank rejestrów:

Reset synchroniczny, pisanie danych synchroniczne, czytanie – dane zawsze dostępne na szynie danych.

Adresy wejściowe muszą być zmapowane na indeksy – no bo adres 32-bitowy, a ja mam tylko 256 rejestrów czyli 8 bitów mi potrzebne. Axi zawsze wystawia adres bajtu. U nas rejestr to 4 bajty. Więc numer rejestru chcemy co cztery. Czyli numer bajtu, który zaczyna dany rejestr. Reg 0 = bajt 0, reg 1 = bajt 4, reg 2 = baj 8 itd. Czyli jeśli chce mieć 256 rejestrów to najwyższy będzie miał adres 255*4= 1020. Czyli 10 bitów potrzeba. Od 9 do 0. No i interesuje nas tak naprawdę modulo z tego przez 4. Więc dwa najmłodsze bity można uciąc tak, żeby było 8 bitów. Czyli ostatecznie trzeba wziąć addr[9:2].

Podpięte, przetestowane – działa nieźle. Trzeba wstawić ss z symulacji.