TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät für Informatik
Institut für Technische Informatik

Beleg zum Komplexpraktikum Prozessorentwurf

Entwurf eines einfachen Mikroprozessors mit dreistufiger Pipeline

Kandetzki Valentin

Matrikelnummer: 3755109
Studiengang: Informationssystemtechnik
Studienjahrgang: 2011

Kemnitz Alexander

Matrikelnummer: 3755118
Studiengang: Informationssystemtechnik
Studienjahrgang: 2011

Dresden, 18. Oktober 2015

Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. Rainer G. Spallek
Verantwortlicher Betreuer: Dr.-Ing. Martin Zabel

Inhaltsverzeichnis

1 Aufgabenstellung

Die Aufgabe bestand darin, einen einfachen Mikroprozessor mit reduzierten MIPS-Befehlssatz zu entwickeln. Dabei war der Prozessor mit Steuerwerk, Datenpfad und 3-stufiger Pipeline sowie der Datenbus mit Wishboneprotokoll in VHDL zu implementieren. Die Komponenten I/O-Controller für UART, ALU sowie Programm- und Datenspeicher waren gegeben. Ziel der Arbeit sollte es sein, ein "Hello WorldProgramm mit Hilfe des Mikroprozessors auf einem FPGA ausführen zu lassen.

1.1 Abarbeitungsreihenfolge

1. Blockdiagramm des Prozessors mit den einzelnen Komponenten und Verbindung zwischen ihnen

2. Implementierung der einzelnen Komponenten in VHDL, testen der Einzelkomponenten

3. Zusammenschalten der Komponenten im Topmodul

4. Test, Simulation und Debugging des Mikroprozessors

5. Portieren des Entwurfes auf ein FPGA und ausführen des Hello World Programms

2 Entwurf

Zu Beginn entwickelten wir mit Hilfe des Buchs Patterson; Henessy: Rechnerorganisation und -entwurfëine Grafische Darstellung der Komponenten des Prozessors. Um diese einfach bearbeiten zu können verwendeten wir das Programm Yed. (Endgültige version siehe Abbildung 1)

Die wichtigsten Bestandteile sind:

2.1 PC (ProgrammCounter)

Einfaches Register, das die Adresse des Aktuellen Befehls anzeigt. Dadurch, dass der Befehl getaktet aus dem Ram geladen wird, muss man i.A. den nächsten PC, den NPC, dafür verwenden. Ausnahme sind Programmstart und angehaltene Pipeline.

2.2 J-adr-calc

Hier wird abhängig vom Sprung typ Die Adresse berechnet. Es gibt drei Typen, die mittels j_dst_control unterschieden werden.
Jump Target: Zieladresse = PC + Target"00".
Jump immediate: Zieladresse = immediate
(mit Übernahme der Obersten Bits der alten Adresse).
Jump register: Zieladresse= Registerinhalt.

2.3 Steuerung

In der Steuerung wird für jeden Befehl entschieden, welche anderen Komponenten des MIPS verwendet werden und wie sie miteinander verbunden sind.

J_dst_controll Bestimmt die Art auf die die Zieladresse eines Sprungs berechnet wird. (J-Typ, R-Typ, I-Typ)

MemRead signalisiert eine Load-Operation

MemWrite signalisiert eine Write-Operation

MemToReg gibt an Welches Ergebnis in ein Register gespeichert wird: Normales ALU-Ergebnis, Daten aus dem RAM oder die Rücksprung-Adresse bei JALR

AluOp legt die Operation der ALU fest

RegWr Signalisiert, ob ein Ergebnis in ein Register geschrieben werden soll.

AluSrc bestimmt, ob die Alu 2 Registerinhalte oder ein Register (rs) mit dem Immediate verrechnet wird.

jumpOp Unterscheidet zwischen den Bedingungen für einen Sprung. Es gibt auch jeweils einen Wert für kein Sprung und unbedingter Sprung.

ShiftSrc dient bei Shift-Operationen zur Unterscheidung zwischen Registerwert und 'Immediate'

Branch Kündigt einen Potientiellen Sprung an, damit der übernächste Befehl eventuell nicht ausgeführt wird.

RegDst Entscheidet ob auf rt oder rd zurückgeschrieben wird

link Legt fest, dass bei JALR die Rücksprungadresse auf R31 (ra) gesichert wird.

Des weiteren gibt es noch die Steuersignal j (in comparator berechnet), das festlegt, ob der Nächste Befehl Sequentiell berechnet wird oder ein Sprungziel ist und pipeline_Enable (im Speicherinterface berechnet), das die Pipeline bei LOAD/STORE-Befehlen stoppt.

2.4 Registers

Enthält die Register 31 Register, wobei R0 fest mit 0 verdrahtet ist. Es können im gleichen Takt 2 Register ausgelesen und 1 Register beschrieben werden, wobei das Lesen in ID-Stufe und das Schreiben in der EX-Stufe eines Befehls ausgeführt wird. Daraus ergibt sich, dass wenn ein Befehl das Ergebnis des letzten Befehls benötigt (read und write Addresse gleich sind), das Ergebnis direkt weitergegeben wird, da das Ergebnis ja erst einen Takt zu spät im Register stehen würde.

2.5 sgn-extend

Dient lediglich zur vorzeichenbehafteten Erweiterung des Immediates auf 32, weil der Immediate 16 Bit groß ist, während die Alu mit 32 Bit rechnet.

2.6 Comparator

Überprüft für bedingte Sprünge, ob die Bedingungen erfüllt sind. Es gibt die Bedingung on Equal, on not Equal, wo Data1 mit Data2 verglichen wird sowie on greater than, less then, reager equal, less equal zero, wo Data1 mit 0 verglichen wird.

2.7 Memory Interface

Verbindet den Prozessor mit RAM und UART. Dabei ist unser Interface Schnittstelle zwischen LOAD/STORE-befehlen und Wishbone-Protokoll. Dieses Interface beinhaltet den Master des Protokolls. Weil die Datenübertragung mehrere Takte benötigt, wird der Pipeline hier solange angehalten bis die Übertragung abgeschlossen ist. Im Fall des RAMs ist das eine feste Zeitspanne in Fall der UART variable Zeit, die durch das Wishboneprotokoll und die angehaltene Pipeline allerdings keinen Einfluss auf den Programmverlauf hat. Im nachfolgenden Bild ist der Zustandsgraf der FSM beigelegt.

2.8 Pipelineregister IF-ID

erstes Pipelineregister

2.9 Pipelineregister ID-EX

zweites Pipelineregister

2.10 ALU

gegeben, Rechenwerk des Prozessors

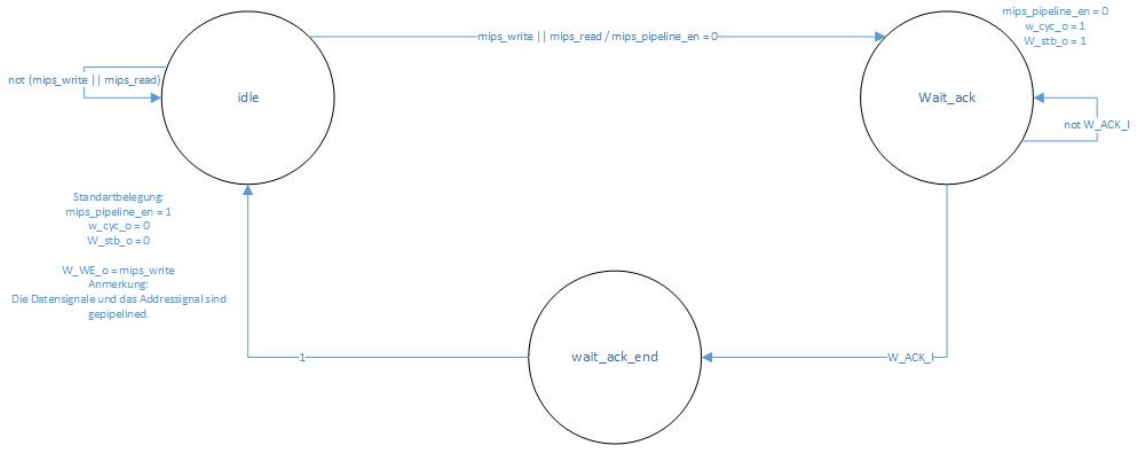

Abbildung 1: FSM_WB_Master

2.11 Operationstypen:

- Rechenbefehl Signalbelegungen: Operation ALU op. Branch, MemRead, MemWrite, link, .. 0. MemToReg 0. Shiftsource nur bei Shift operation abhängig Register/'Immediate'. RegDst und AluSrc abhäning von Imm/2 Register
- Immediate
- 2 Register
- Load/Store
- Sprungbefehl

Sprungbefehle führen in einer 3-Stufigen Pipeline bereits zu einem Harzad. Deswegen wird der Befehl 2 nach dem Sprungbefehl durch eine NOP ersetzt (auch wenn der Sprung am Ende nciht ausgeführt wird). Dafür ist das Signal Branch zuständigk, das einen Potentiellen Sprung markiert.
Unterteilung in Bedingung (in comporator geprüft) zuständiges
Signal: jumpOp
(-kein Sprung)
- unbedingt
- bedingt
- Vergleich zweier Registerinhalte
- auf Gleichheit
- auf Ungleichheit
- Vergleich eines Registerinhalts mit 0
- größer
- kleiner
- größer oder gleich
- kleiner oder gleich

und Typ der Zieladresse (J-adr-calc geprüft) : zuständiges
Signal $\mathrm{J}_d st_c ontrol$
$-$ $Jump(target)$
$-$ $JumpRegister$
$-$ $JumpImmediate$

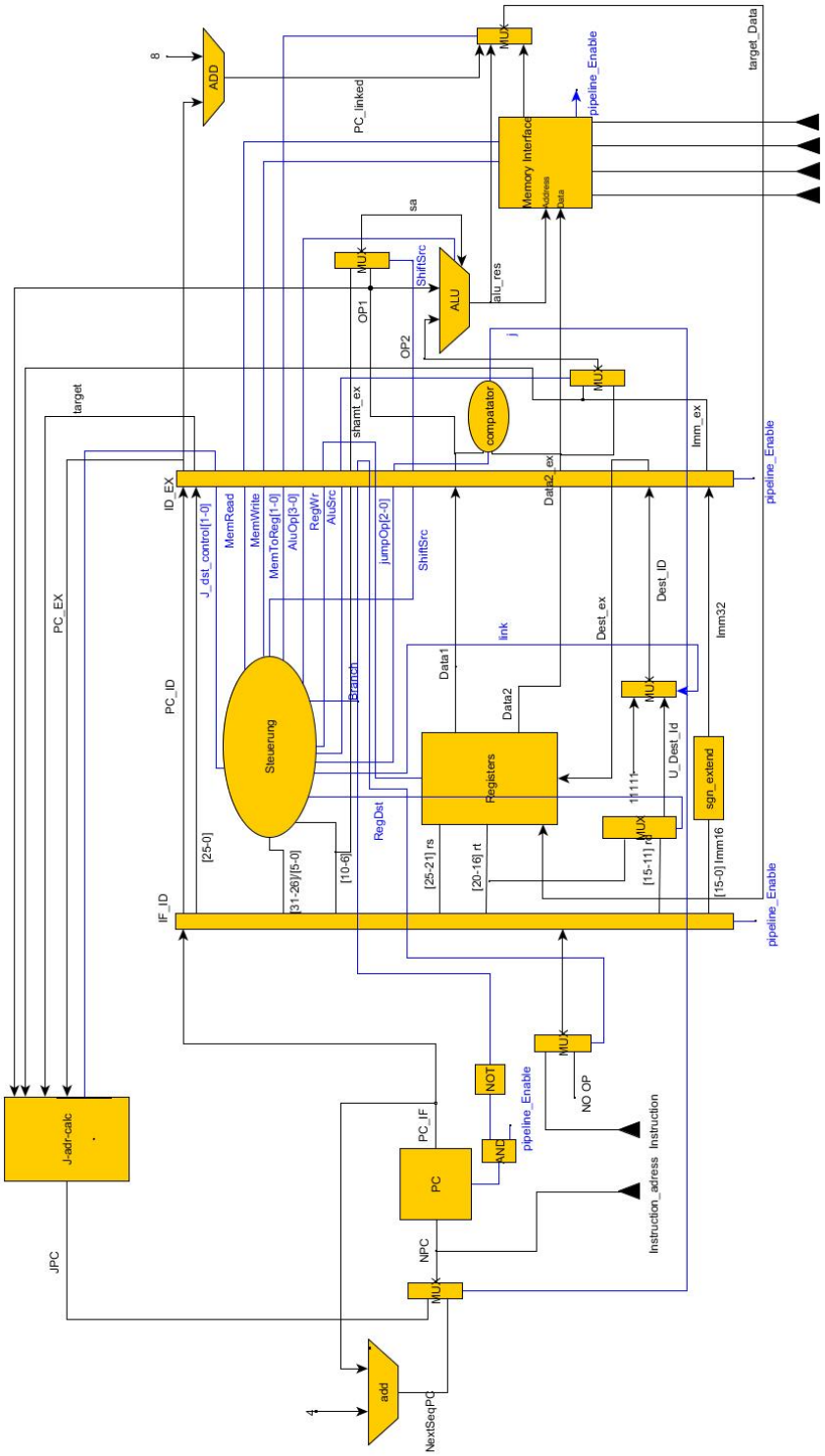

Abbildung 2: Komponenten

3 Test

3.1 Comparator

Es gibt je 4 Testfälle zu den Operation branch on equal, branch on not equal, no jump. Bei branch on equal / branch on not equal werden im ersten Fall zwei gleiche Zahlen miteinander verglichen (zwei Nullen). Im zweiten Fall zwei ungleiche Zahlen und im dritten zwei gleiche Zahlen ungleich null. Das Ergebnis soll bei branch on not equal im Fall eins „0“ sein, im Fall zwei „1“ und im Fall drei „0“. Die Ergebnisse zu Branch on equal sind entsprechend invers. Im Testfall no Jump werden Operationen getestet, in denen kein Sprung ausgeführt werden soll.

3.2 J_adr_calc

Im Test des J_adr_calc werden verschiedene Werte („00“, „01“, „10“) für J_dst_control verwendet und geprüft, ob die Richtige Sprungadresse berechnet wurde. Registers

3.3 Steuerung

In der Testbench der Steuerung werden die Ergebnisse der Ausgangssignale für die Einzelnen Befehle (Operationen) getestet. Bei der Operation“000000“ wird hierbei „funct“ genauer betrachtet.
funct ¡= ”000000- SLL
funct ¡= ”000010- SLR
funct ¡= ”000011- SRA
funct ¡= ”000100- SLLV
funct ¡= ”000110- SLRV
funct ¡= ”000111- SRAV
funct ¡= ”100000- ADD
funct ¡= ”100001- ADDU
funct ¡= ”100010- SUB
funct ¡= ”100011- SUBU
funct ¡= ”100100- AND
funct ¡= ”100101- OR
funct ¡= ”100110- XOR
funct ¡= ”100111- NOR
funct ¡= ”101010- SLT
funct ¡= ”101011- SLTU
funct ¡= ”001000- JR
funct ¡= ”001001- JALR

operation ¡= ”000010- J
operation ¡= ”000011- JAL
operation ¡= ”000100- BEQ
operation ¡= ”000101- BNE
operation ¡= ”001000- ADDI
operation ¡= ”001001- ADDIU
operation ¡= ”001010- SLTI
operation ¡= ”001011- SLTIU
operation ¡= ”001100- ANDI
operation ¡= ”001101- ORI
operation ¡= ”001110- XORI
operation ¡= ”001111- LUI
operation ¡= ”100011- LW
operation ¡= ”101011- SW

Die Erwarteten Ausgangssignale können dem Quellcode entnommen werden.

3.4 Wishbone_Master

Für den Wishbone Master werden drei Testfälle unterschieden. (Store Word, Load Word und read and write). Dabei soll bei Store Word auf denRam geschrieben werden und bei Load Word von dort Geladen werden. Bei read and write tritt ein Fehler auf.

3.5 mips_top

Hier wir die Zusammenschaltung der einzelnen Komponenten getestet. Dabei werde direkte Assemblerbefehle wie Add, Sub, Or, Jump getestet und geschaut ob die Registerinhalte dem richtigen Ergebnis entsprechen.

3.6 Megatop

Hier werden ebenfalls Assemblerbefehle getestet. Da der bootram hier mit enthalten ist können auch die Load- und Stoarebefehle getestet werden.

World Waveforms.png

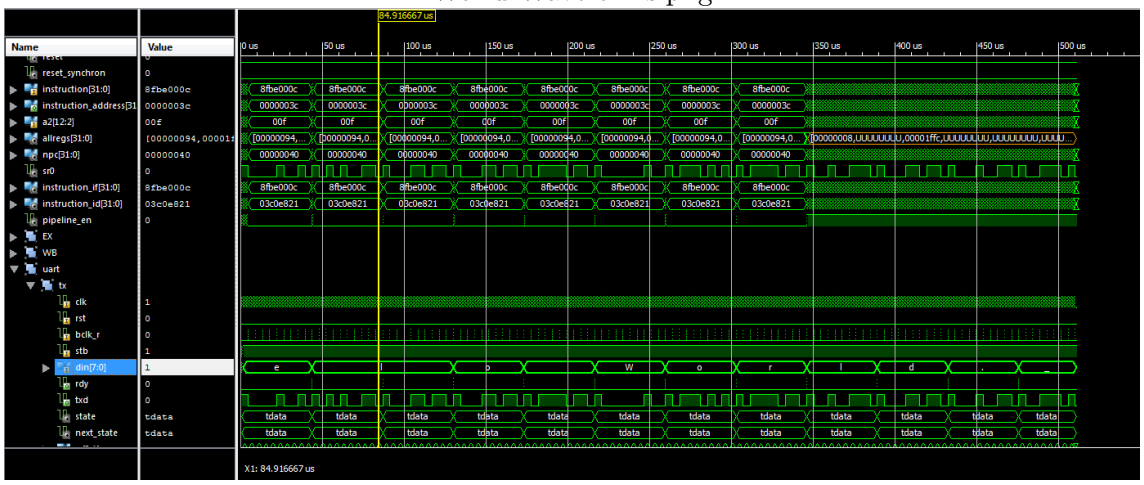

Abbildung 3: Hello World Ausgabe

condition false Waveforms.png

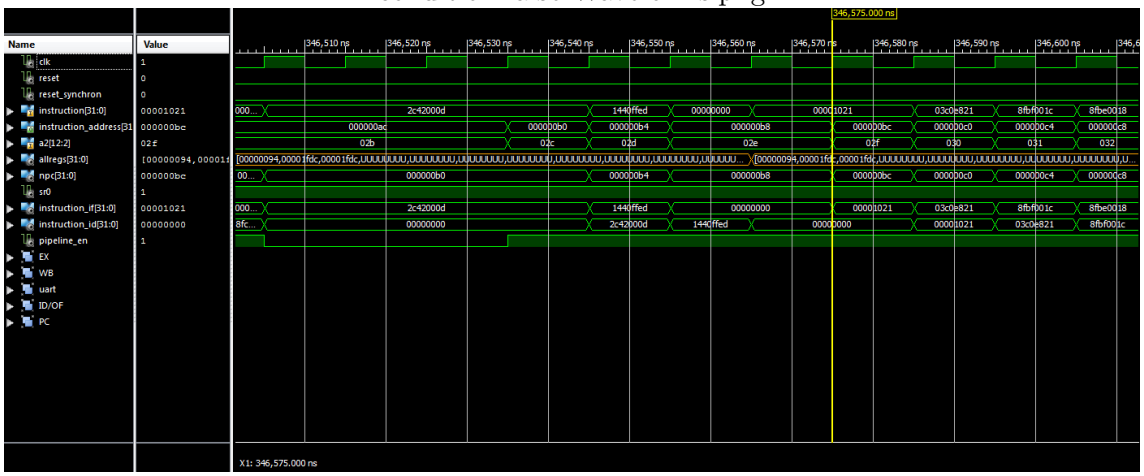

Abbildung 4: Branch condition false Waveforms

Waveforms.png

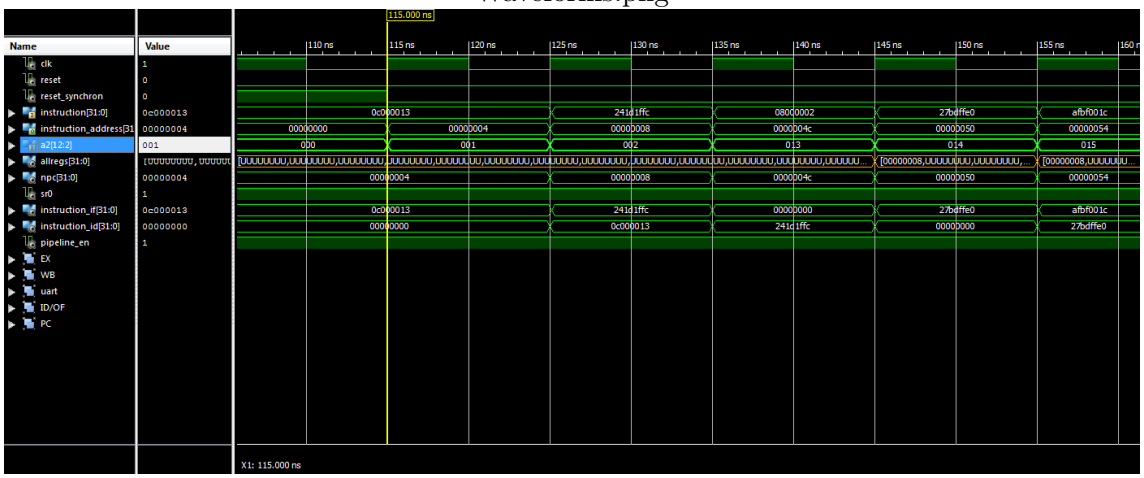

Abbildung 5: Start Waveforms

handshake waveforms.png

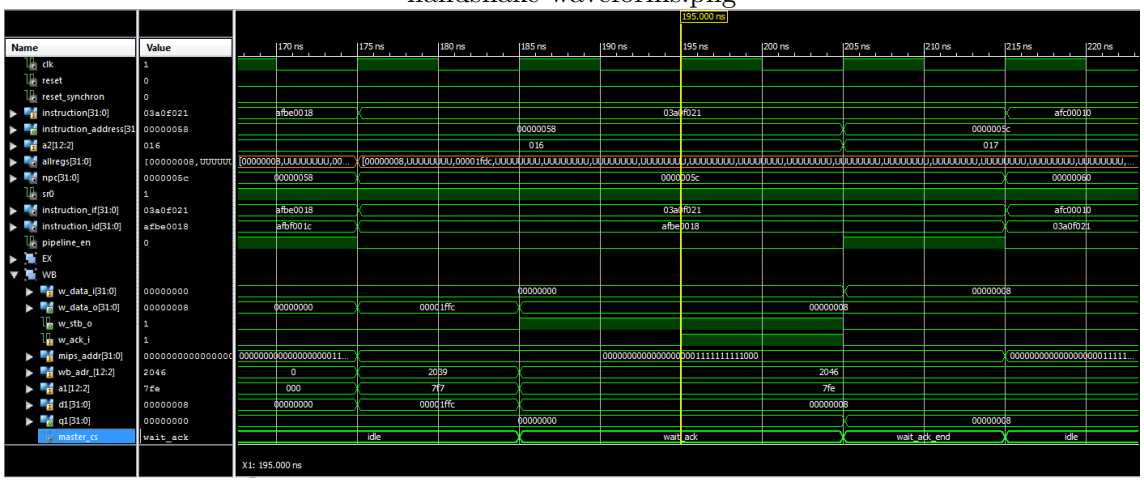

Abbildung 6: wishbone handshake waveforms

handshake LOAD waveforms.png

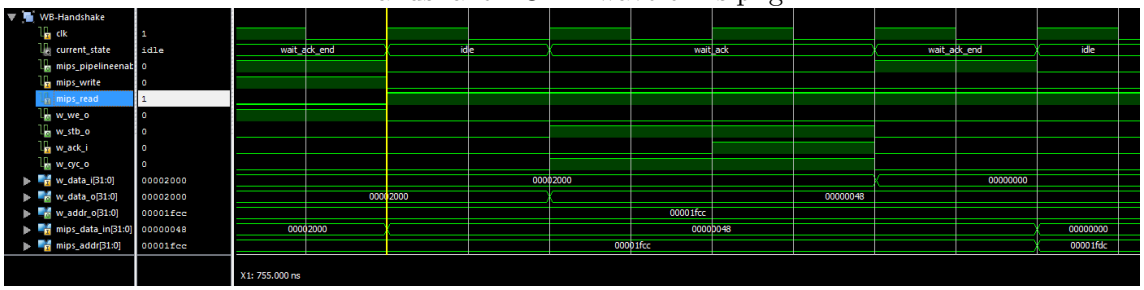

Abbildung 7: wishbone handshake LOAD waveforms

handshake STORE waveforms.png

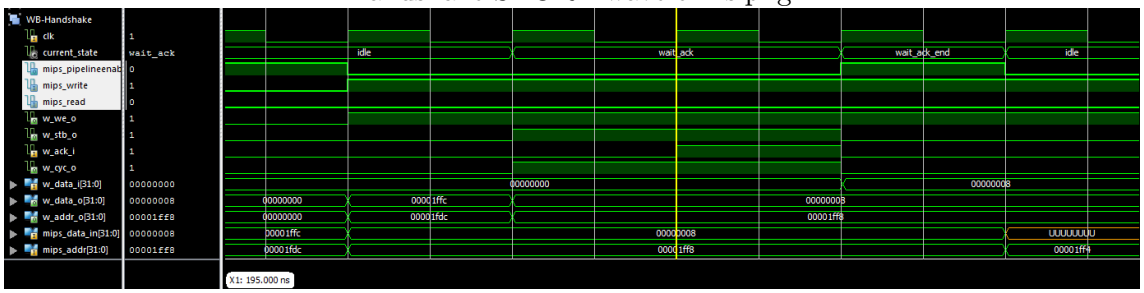

Abbildung 8: wishbone handshake STORE waveforms

Abbildungsverzeichnis