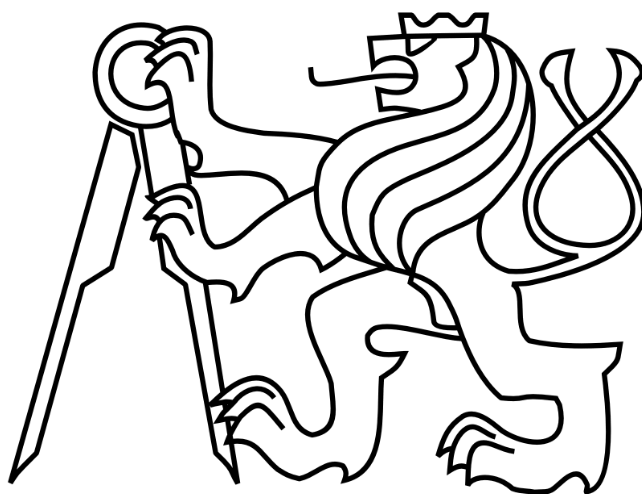
České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra teorie obvodů



DIPLOMOVÁ PRÁCE

Hardwarový akcelerátor pro BCI aplikace

Praha, 2014

Diplomant: Bc. Vladimír Beran
Vedoucí: Ing. Jakub Šťastný, Ph. D

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Vladimír Beran**

Studijní program: Otevřená informatika (magisterský)
Obor: Počítačové inženýrství

Název tématu: **Hardwarový akcelerátor pro rozhraní mozek-stroj**

Pokyny pro vypracování:

Navrhněte a implementujte systém pro hardwarovou akceleraci výpočtů v oblasti BCI založený na FPGA obvodu.

1. Seznamte se s problematikou BCI systémů a s řešením [1].
2. Analyzujte možnosti implementace na systémové úrovni, proveďte systémový návrh, zvažte alternativy řešení (CPU vs systém s dedikovanými moduly). Navrhněte integraci FPGA akcelerátoru do řešení [1].
3. Zpracujte systémový návrh základního bloku – výpočetní dlaždice založený na [2], zvažte alternativní komunikační rozhraní a protokoly mezi dlaždicemi s ohledem na jednoduchost návrhu, znovupoužitelnost a snížení spotřeby elektrické energie. Navrhněte případná vylepšení konceptu [2] na blokové i systémové úrovni s ohledem na tato kriteria; zachovejte kompatibilitu s existujícím SW systémem a modularitu.
4. Implementujte generický blok a další důležité bloky BCI systému na RTL úrovni v jazyce VHDL, demonstrujte správnou funkci. Jako demonstrační blok zvolte některý z bloků v BCI systému.

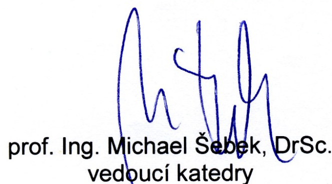
Seznam odborné literatury:

[1] J. Šťastný, J. Doležal, V. Černý. Real Time BCI Processing Software - System Specification. Katedra teorie obvodů FEL ČVUT, FPGA Laboratoř. 2012

[2] J. Šťastný. A Modular Hardware Platform for Brain Computer Interface. In Applied Electronics, Applied Electronics, pp 287-290, September 2012. ISBN 978-80-261-0038-6

Vedoucí: Ing. Jakub Šťastný, Ph.D.

Platnost zadání: do konce letního semestru 2013/2014



prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 18. 12. 2012

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, dne ………………………. ……………………………………..

Podpis

Poděkování:

Tímto bych rád poděkoval vedoucímu diplomové práce Ing. Jakubu Šťastnému, Ph. D za vedení a rady při vypracování. Dále Ing. Petru Pavlatovi a Ing. Vladimíru Černému za cenné rady ohledně systému BCI a Mgr. Adéle Czimprichové za pomoc s úpravou obrázků.

Abstrakt

Předmětem této práce je systémový návrh a implementace IP maker na FPGA umožňujících HW akceleraci pro BCI výpočty. Tato IP makra musí být kompatibilní se stávajícím SW a HW. Při návrhu maker je kladen důraz na nízkou spotřebu.

V první části práce jsou probírány techniky pro snížení spotřeby digitálních obvodů nejen na FPGA. V druhé části je proveden systémový návrh celé HW koncepce BCI. Z něho vyplynuly požadavky na jednotlivá IP makra. V poslední části je popsána jejich implementace a verifikace.

Abstract

System design and implementation of IP macros FGPA enabling HW acceleration for BCI calculation is the scope of this theses. This IP macros must be compatible with existing SW and HW. When designing the macros the emphasis is on low power consumption.

Techniques to reduce power consumption of digital circuits not only FPGA are discused in the first part theses. The entire system design concept HW BCI is executed in the second part. It also made clear requirements for each macro. Their implementation and verification is described in the last section.

Obsah

1Úvod..1
2BCI systém 2
2.1Dosažené výsledky...3
2.2Cíle práce 4
3Základní techniky návrhu s nízkou spotřebou..5
3.1Příkon CMOS obvodů:...5
3.1.1Spínací část příkonu:..5
3.1.2Zkratová část příkonu: 5
3.1.3Svodové proudy (leakage)...6
3.2Optimalizace na technologické úrovni:..7
3.2.1Minimalizace spínaných kapacit..8
3.3Optimalizace na logické a RTL úrovni...9
3.3.1Snížení špičkové spotřeby 9
3.3.2Vliv sdíleni prostředků na spotřebu...9
3.3.3Vliv pořadí operací na spotřebu...12
3.3.4Hradlování hodin a vypínání části obvodu...12
3.3.5Redukce hazardů 13
3.4Optimalizace na architektonické úrovni 13
3.4.1Paralelismus a pipelining...14
3.4.2Volba číselné reprezentace...14
3.4.3Přístup k pamětem...15
3.4.4Vliv jednotlivých opatření na spotřebu 16
3.5Optimalizace na systémové úrovni:..18
3.5.1Výběr architektury..19
3.5.2Výběr algoritmu, dekompozice mezi HW a SW...19
3.6Shrnutí technik pro snížení spotřeby 20
4Systémový návrh hardwarové platformy BCI...21
4.1Typická práce systému...21
4.2Volba architektury..22
4.3Integrace s existujícím HW a SW...23
4.3.1Sběrnice Wishbone..24
4.4Proudové zpracování dat..26
4.4.1Bloky DSP...27
4.4.2Bloky GIB..27
4.4.3Další typy bloků v proudovém zpracování dat...29
4.5Hraniční bloky...29
4.6Minimalizace spotřeby 30
4.6.1Odhad velikosti RAM v bloku GIB..30
4.7Koncept ovládacího SW...32
4.7.1Neúplné a chybějící pakety...33
5Návrh bloku GIB...34
5.1Popis rozhraní..34
5.1.1Rozhraní pro zápis do bloku GIB...34
5.1.2Rozhraní pro čtení z bloku GIB...35
5.2Vnitřní uspořádání...36
5.2.1Řadič...37
5.3Verifikace bloku GIB..38
5.3.1RTL simulace řadiče a datové cesty...38
5.3.2Hradlová simulace bloku GIB..38

5.3.3Hradlová simulace bloků GIB v sérii s dummy bloky a hraničními bloky......................38
5.4Implementace bloku GIB...39
5.5Návrh dummy bloku...39
5.5.1Vnitřní uspořádání bloku dummy...39
5.6Výsledky verifikací bloků GIB a dummy..40
6Návrh mostu mezi Wishbone a GIB..41
6.1Vnitřní uspořádání hraničních bloků..41
6.1.1Blok bridge master..41
6.1.2Blok bridge slave...42
6.1.3Pole registrů 43
6.2Verifikace hraničních bloků...45
6.2.1RTL simulace řadiče, datové cesty a pole registrů..45
6.2.2Hradlová simulace obou mostů...46
6.2.3Hradlová simulace obou mostů s připojením ke zbytku výpočetní posloupnosti............46
6.3Implementace...46
6.3.1Varování..47
6.4Výsledky verifikace 47
7Připojení výpočetní posloupnosti k procesoru OpenRISC...48
7.1Konfigurace arbitru..48
7.2Sestavení testovací konfigurace..48
7.3Procesor z pohledu programátora 50
7.4GNU nástroje..50
7.5Testovací program 51
8Výpočetní modul 52
8.1Návrh programovatelného DSP bloku..52
8.1.1Registrové pole..53
8.1.2PC..53
8.1.3ROM..53
8.1.4Jednotlivé příkazy..53
8.2verifikace..55
8.3Návrh filtru...56
8.3.1Verifikace filtru...57
8.3.2Implementace 57
8.3.3Výsledky verifikace...58
8.4Implementace...59
8.4.1Varování..60
8.5Výsledky verifikace..60
9Závěr...61
10Použitá literatura a zdroje 62
11Slovník zkratek 64
12Přílohy...65
13Obsah přiloženého CD...66

Seznam obrázků

Obrázek 1: Schéma systému BCI. Převzato z [4]..........2
Obrázek 2: Hraní hry Arkanoid ovládané přes BCI. Převzato z [6]..........4
Obrázek 3: Základní složky svodových proudů..........7
Obrázek 4: Zpoždění hradla v závislosti na napájecím napětí. Převzato z [1]..........7
Obrázek 5: Efekt snižování prahového napětí na zpoždění při různých napájecích napětích. Převzato z [1]..........8
Obrázek 6: Kompromis mezi dynamickými a leakage ztrátami. Převzato z [1]..........8
Obrázek 7: Spotřeba energie ku poměru W/L pro různá α. Převzato z [1]..........8
Obrázek 8: Rozprostření spotřeby. Převzato z [2]..........9
Obrázek 9: Nesdílená sběrnice. Převzato z [1]..........10
Obrázek 10: Sdílená sběrnice. Převzato z [1]..........10
Obrázek 11: Porovnání spínací aktivity pro sdílenou a nesdílenou sběrnici pro čítač. Převzato z [1]. 10
Obrázek 12: Pravděpodobnost přechodu pro paralelní a časově multiplexovanou sčítačku. Převzato z [1]..........11
Obrázek 13: Redukování aktivity reorganizací vstupních signálů. Převzato z [1]..........12
Obrázek 14: Ukázka hradlování hodin pro snížení spotřeby. Převzato z [1]..........12
Obrázek 15: Redukce šíření hazardů. Převzato z [2]..........13
Obrázek 16: Rozdělení datové cesty za pomoci paralelismu. Převzato z [1]..........14
Obrázek 17: Jednoduchá datová cesta. Převzato z [1]..........14
Obrázek 18: Rozdělení datové cesty za pomoci pipeline. Převzato z [1]..........14
Obrázek 19: Pravděpodobnost přechodu jednotlivých bitů vůči stupni korelace dat - dvojkový doplněk. Převzato z [1]..........15
Obrázek 20: Pravděpodobnost přechodu jednotlivých bitů vůči stupni korelace dat – znaménkové rozšíření. Převzato z [1]. 15
Obrázek 21: Rozdílné přístupy do paměti. Převzato z [1]..........15
Obrázek 22: Vliv čtení a zápisu na spotřebu při různých volbách adresního kódování a různé délce slova. Převzato z [2]..........16
Obrázek 23: Vliv pořadí zápisů a čtení na spotřebu při různých adresních vzdálenostech a různé velikosti paměti. Převzato z [2]..........16
Obrázek 24: Porovnání spotřeby u volatilních a nevolatilních FPGA. Převzato z [2]..........19
Obrázek 25: Spojení HW a SW části. Převzato z [3]..........21
Obrázek 26: Schéma bloku GIB. Převzato z [3]..........21
Obrázek 27: Architektura OpenRISC OR1200. Převzato z [7]..........23
Obrázek 28: Arbitr typu Crossbar switch. Převzato z [11]..........24
Obrázek 29: Typický průběh čtení na sběrnici Wishone.Převzato z [11]..........25
Obrázek 30: Typický průběh zápisu na sběrnici Wishbone.Převzato z [11]..........25
Obrázek 31: Schéma zapojení proudového zpracování dat..........26
Obrázek 32: Princip funkce bloku GIB..........28
Obrázek 33: Proudové zpracování dat s rozvětvením..........29
Obrázek 34: Korespondeční signály pro zápis do RAM..........34
Obrázek 35: Korespondenční signály pro čtení z RAM..........35
Ilustrace 36: Schéma vnitřního uspořádání bloku GIB..........36
Obrázek 37: Schéma stavového automatu v bloku GIB..........37
Obrázek 38: Schéma stavového automatu bloku dummy..........40
Obrázek 39: Schéma datové cesty bloku dummy..........40
Obrázek 40: Schéma stavového automatu v bloku bridge master..........41
Obrázek 41: Vnitřní schéma bloku bridge master..........42
Obrázek 42: Schéma stavového automatu v bloku bridge slave..........42

Obrázek 43: Vnitřní schéma bloku bridge slave..43
Obrázek 44: Vnitřní uspořádání bloku register_field..45
Obrázek 45: Ideové schéma připojení debug jednotky. Převzato z [14]...49
Obrázek 46: Ideové schéma zapojení testovací konfigurace...50
Obrázek 47: Ideové schéma programovatelného DSP bloku...53

Seznam tabulek

Tabulka 1: Sdílení prostředků. Převzato z [12].....11
Tabulka 2: Vliv architektury na snižování spotřeby. Převzato z [1].....14
Tabulka 3: Optimalizace spotřeby pamětí. Převzato z [2].....16
Tabulka 4: Optimalizace spotřeby I/O. Převzato z [2].....17
Tabulka 5: Optimalizace spotřeby hodinového stromu. Převzato z [2].....17
Tabulka 6: Optimalizace spotřeby logiky. Převzato z [2].....18
Tabulka 7: Srovnání základních vlastností CPU/DSP a FPGA.....22
Tabulka 8: Signály na sběrnici Wishbone.....25
Tabulka 9: Struktura RTP paketu. Převzato z [8].....30
Tabulka 10: Struktura hlavičky EEG a EEG dat v RTP paketu. Převzato z [8].....31
Tabulka 11: Struktura dat posílaných do bloku GIB. Převzato z [8].....32
Tabulka 12: Řešení problému s RTP pakety.....33
Tabulka 13: Rozhraní pro zápis do bloku GIB.....35
Tabulka 14: Rozhraní pro čtení z bloku GIB.....35
Tabulka 15: Použitá HDL makra v GIB.....39
Tabulka 16: Popis registrů v poli registrů.....43
Tabulka 17: Využití Spartanu 6 (xc6slx45-3csg324) výpočetní posloupností.....46
Tabulka 18: Mapování komponent připojených na Wishbone.....48
Tabulka 19: Popis kódu jednotlivých instrukcí.....54
Tabulka 20: detaily implementace demonstrační DSP funkce.....58
Tabulka 21: Výsledky HDL syntézy programovatelného DSP bloku.....59

1 Úvod

BCI (*Brain computer interface*) je systém, který je na základě klasifikace EEG (*Elektroencefalogram*) schopen rozpoznat mentální aktivitu uživatele. BCI může sloužit například k ovládání pohybu kurzoru na PC nebo k ovládání kolečkového křesla. Tento systém je schopen rozeznat otevřené nebo zavřené oči, představy pohybu levé nebo pravé ruky, představy pohybu prstů a jazyka. Na katedře teorie obvodů je v rámci FPGA (*Field programable gate array*) laboratoře vyvíjen právě takovýto systém. Fakultní BCI má být výhledově schopno rozeznávat ještě více aktivit.

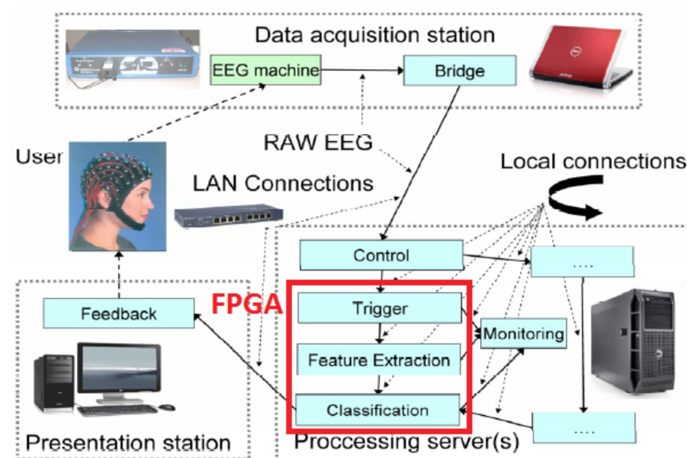
Zpracování a klasifikace EEG vyžadující např. skryté Markovské modely, slepou separaci nebo neuronové sítě je velmi výpočetně náročná. BCI je testováno on-line na skutečných uživatelích, jednotlivé algoritmy jsou testovány off-line na předem nahraných datech. V obou případech bylo EEG zpracováváno na klasických PC. Klasické stolní PC (ale i notebook) má však velké rozměry a zejména velkou spotřebu. PC je špatně transportovatelné a je obtížné jej napájet z baterie. Pro zpracování EEG on-line u uživatelů je potřeba se poohlédnout po jiném energeticky úspornějším a na objem menším zařízení. Právě takové zařízení se snaží vyvinout na katedře teorie obvodů. Dostatečně malé zařízení, které bude moci být napájeno například z baterie a uživatel je bude moci mít neustále při sobě a zároveň bude zvládat zpracování EEG v reálném čase.

Tato práce se bude zabývat návrhem hardwarové akcelerace na obvodech FPGA, kdy hlavní myšlenka je taková, že díky paralelismu,který obvody FPGA umožňují, bude možné celou nebo významnou část klasifikace EEG provádět v reálném čase na jediném čipu FPGA.

2 BCI systém

BCI systém vyvíjený na katedře má sloužit k ovládání externího zařízení člověkem. Současné BCI nejsou schopny pracovat s vyšším datovým tokem než je cca. 100 bitů za minutu, jsou schopné rozeznávat představu pohybu levé/pravé ruky, jsou schopny detekovat představy pohyby prstů a jazyka. Pro dosažení vyšší komunikační rychlosti je nutno rozeznávat více mentálních stavů a používat výpočetně náročnější metody. Právě návrhem hardwarové akcelerace pro BCI zařízení, které bude schopno rozpoznávat EEG s vysokým rozlišením (*high resolution EEG recognition*), se zabývá tato práce. Největším problémem, se kterým se BCI potýká, je malý odstup signál-šum [3].

Zařízení BCI vyvíjené na katedře snímá EEG na vzorkovací frekvenci 256 - 1024 Hz, 16 bitů na vzorek z 2 - 45 elektrod (v závislosti na konfiguraci), které jsou umístěny na povrchu hlavy. Celé BCI vyvíjené na katedře je navrženo jako modulární, kdy mezi sebou jednotlivé bloky komunikují prostřednictvím RTP (*Real-time Transport Protocol*) buďto po síti Ethernet nebo i interně mezi jednotlivými moduly např. v rámci jednoho PC. Hlavní výhodou takového řešení je modularita, kdy mohou být jednotlivé bloky lehce zaměňovány [3].



Obrázek 1: Schéma systému BCI. Převzato z [4]

Na obrázku 1 je potom blokové schéma celého BCI. Jednotlivé bloky jsou popsány níže [4].

Data acquisition station

Skládá se z nahrávacího zařízení EEG (*EEG machine*) a mostu (*bridge*). Nahrávací zařízení EEG nahrává data a distribuuje je v proprietárním formátu (proprietární UDP *User datagram Protocol* formát). Kvůli nezávislosti zbytku BCI na proprietárním formátu je použit datový most. Tento most převádí proprietární formát paketů na RTP pakety, se kterými pracuje zbytek BCI. V současnosti je BCI schopno pracovat s encefalografy *Alien* nebo *BIOPAC*. Most běží na samostatném PC a je naprogramován v jazyku JAVA.

Processing server(s)

Zde probíhá zpracování a klasifikace EEG. Skládá se z několika modulů napsaných v jazyce JAVA, které spolu komunikují prostřednictvím posílání RTP paketů. Může běžet na jednom nebo více počítačích.

Modul *Control*

Slouží pro řízení běžícího systému v reálném čase. Uživatel může pomocí jednoduchého grafického rozhraní ovládat a nastavovat další moduly pomocí specializovaných RTP paketů. V tomto modulu se například nastavuje, zda-li se jedná o testovací nebo trénovací množinu dat.

Modul *Trigger*

Slouží pro časovou synchronizaci. Generuje sekvenci stavů na základě předdefinovaného experimentálního protokolu.

Modul *Feature Extraction*

Zajišťuje filtraci signálů za pomoci filtrů FIR (*Finite impulse response*), odstraňuje rušivé frekvence (50 Hz apod). Počítá odhad výkonu za pomoci integračního filtru.

Modul *Classification*

Provádí klasifikaci naměřených dat například za pomoci jednoduché jednovrstvé sítě perceptronů.

Modul *Monitoring*

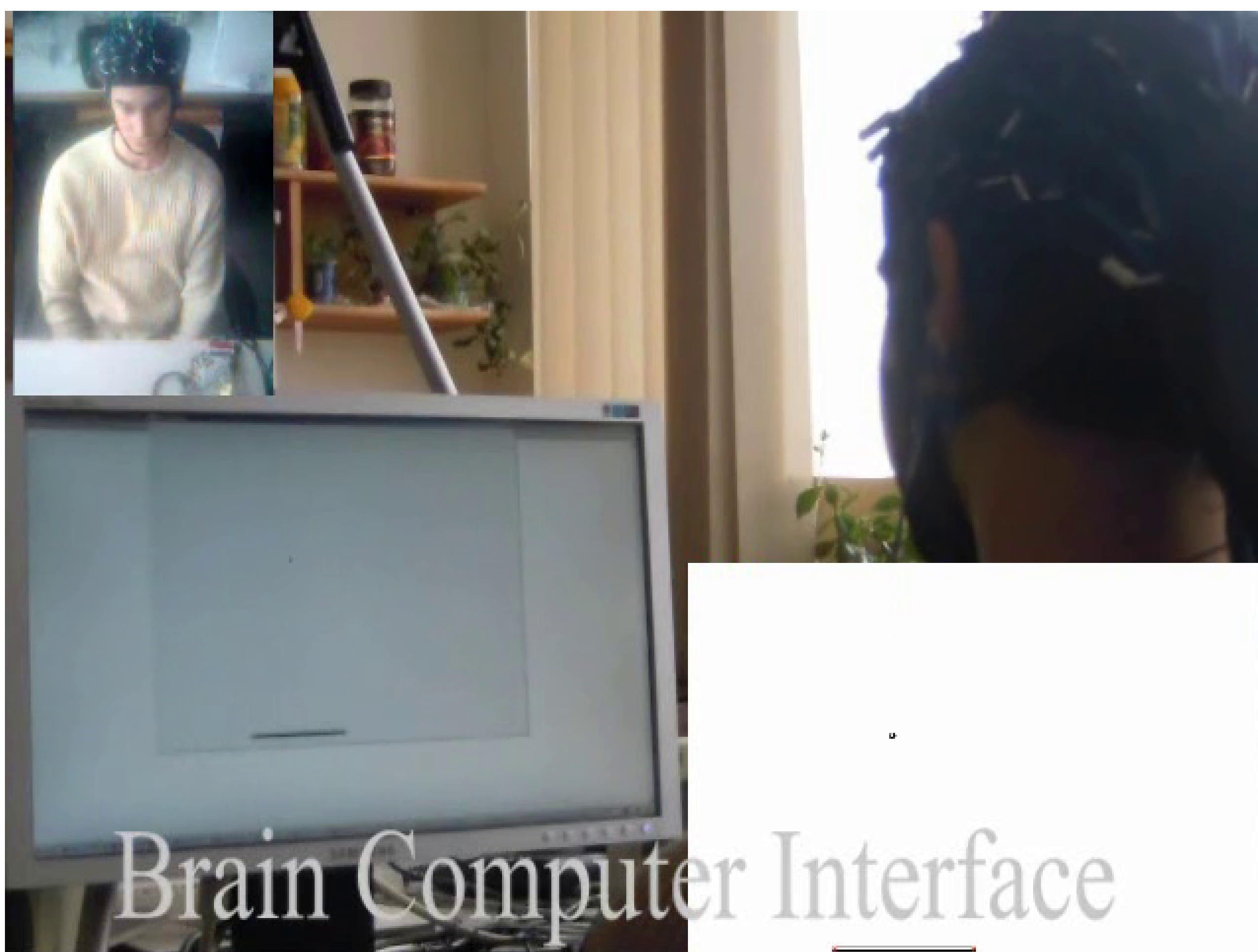
Zobrazuje nastavení experimentu a je schopen zachytit veškerou komunikaci mezi ostatními moduly.

Presentation station

Stanice zajišťující zpětnou vazbu mezi BCI a uživatelem. Tuto zpětnou vazbu může tvořit například hraní hry Arkanoid nebo při detekci levá/pravá ruka směr šipky podle detekované ruky. Modul zajišťující zpětnou vazbu je naprogramován v jazyce JAVA.

2.1 Dosažené výsledky

Všechny pokusy s fakultním BCI jsou dělány v obyčejné nestíněné kanceláři. BCI je prozatím schopno rozeznat představu pohybu levé nebo pravé ruky. Za pomoci BCI je možno ovládat v reálném čase jednoduchou hru Arkanoid, jak je ukázáno na obrázku 2. Úspěšnost při hraní této hry se u některých jedinců pohybuje okolo 70% [4].



Obrázek 2: Hraní hry Arkanoid ovládané přes BCI. Převzato z [6].

2.2 Cíle práce

Požadavky na vysoký výpočetní výkon a nízkou spotřebu celého zařízení vedou na paralelní a distribuované zpracování signálů. Oba tyto požadavky mohou být splněny při použití FPGA. Nízké spotřebě může dopomoci vhodný výběr FPGA obvodu (typ s antipojistkami nebo s pamětí flash). Pro implementaci na FPGA by byly vhodné zejména bloky, které jsou v obrázku 1 vyznačeny červeným obdélníkem. Takovéto FPGA by alespoň během vývoje mělo být schopné pracovat s RTP, se kterým pracuje zbytek BCI systému (*Data acquisition station* a *Presentation station*). Příjem a odesílání RTP paketů již bylo vyřešeno v rámci diplomové práce [5]. Dalším úkolem je tedy přemístění červeně označených bloků v obrázku na FPGA. Avšak i v rámci FPGA by bylo vhodné zejména kvůli snadnější implementaci a ladění zachovat určitou míru modularity. Právě návrh hardwarové akcelerace zachovávající modularitu BCI systému, implementace, verifikace a připojení k již existujícímu HW je cílem této práce.

3 Základní techniky návrhu s nízkou spotřebou

Jak bylo naznačeno v předcházejících kapitolách, BCI je výpočetně a energeticky náročný systém. Pokud ho má mít uživatel neustále při sobě, bude nutné ho napájet z baterie. Protože kapacita baterií se v posledních letech příliš nezvedla, je nutno snížit co nejvíce spotřebu BCI systému. V následujících kapitolách budou ukázány techniky, které mohou přispět ke snížení spotřeby. Nejprve se zaměřím na vlivy ovlivňující spotřebu v obecných CMOS (*Complementary Metal–Oxide–Semiconductor*) obvodech a následně se budu věnovat konkrétním doporučením pro snížení spotřeby, a to z pohledu jednotlivých úrovní návrhu. Nakonec budou shrnuty techniky použitelné na FPGA.

3.1 Příkon CMOS obvodů:

Celkový příkon CMOS obvodů shrnuje následující rovnice :

$$P_{avg} = P_{spínání} + P_{zkrat} + P_{leakage} = \alpha_{0\rightarrow 1} C_L \cdot V_{dd}^2 \cdot f_{clk} + I_{sc} \cdot V_{dd} + I_{leakage} \cdot V_{dd} \quad (1)$$

přičemž $P_{spínání} + P_{zkrat}$ tvoří dynamickou část příkonu a $P_{leakage}$ tvoří statickou část příkonu[1].

3.1.1 Spínací část příkonu:

Příkon pro spínací část je určen opakovaným nabíjením kapacit hradel a propojení a je definován jako

$$P_{spínání} = \alpha_{0\rightarrow 1} \cdot C_L \cdot V_{dd}^2 \cdot f_{clk} \quad (2)$$

kde, $\alpha_{0\rightarrow 1}$ je pravděpodobnost , že v každém hodinovém cyklu uzel s kapacitou C_L přejde ze stavu 0 do stavu 1. V_{dd} je napájecí napětí obvodu a f_{clk} je frekvence hodin. Výraz $\alpha_{0\rightarrow 1} C_L$ vyjadřuje efektivní kapacitu C_{ef} . Průměrná energie přechodů v jednom hodinovém taktu se dá vyjádřit jako $C_{ef} \cdot V_{dd}^2$. Jednou z technik snížení spotřeby CMOS obvodů je kromě fyzického snížení počtu hradel a propojení také zmenšení této efektivní kapacity, a to jak statisticky při snížení pravděpodobnosti přechodu vhodnou volbou logické funkce, tak dynamicky snížením počtu falešných přechodů (hazardů) [1].

3.1.2 Zkratová část příkonu:

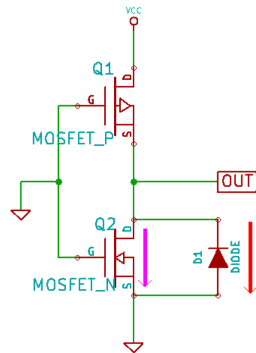
Zkratovou část příkonu lze vyjádřit jako

$$P_{zkrat} = I_{SC} \cdot V_{dd} \quad (3)$$

kde I_{SC} je zkratový proud, který prochází hradlem pokud jsou oba tranzistory otevřeny současně. Ve skutečných obvodech CMOS nejsou totiž náběžné a sestupné hrany nekonečně rychlé a nastává pak situace, kdy dojde na krátký okamžik k současnému otevření PMOS a NMOS tranzistorů. Minimalizaci těchto ztrát je možno provést vhodným nastavením prahových napětí tranzistorů vůči napájecímu napětí. Tato prahová napětí musí splňovat : $V_{Tn} < V_{\text{in}} < V_{dd} - |V_{Tp}|$ kde V_{Tp} a V_{Tn} jsou prahová napětí PMOS a NMOS tranzistorů. Dále je vhodné, aby vzestupné a sestupné hrany hodin trvaly stejnou dobu [1].

3.1.3 Svodové proudy (*leakage*)

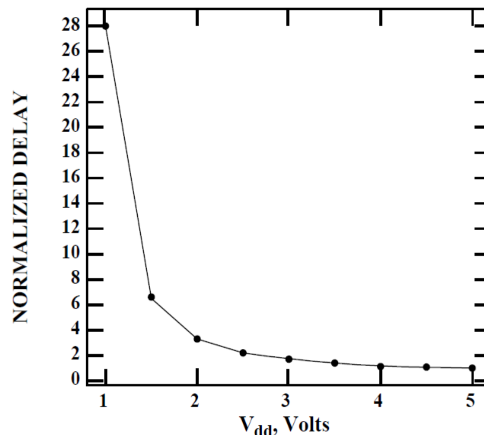
Existují dvě základní složky svodových proudů (*leakage*), a to závěrný diodový svodový proud (*reverse-bias diode leakage*) a podprahový svodový proud (*subtreshold leakage*). Závěrný diodový svodový proud způsobují parazitní diody obsažené v MOS tranzistorech. Pokud je jeden z tranzistorů otevřený a druhý zavřený vůči společnému potenciálu na *drainu*, tak parazitní diodou tranzistoru prochází proud v závěrném směru. Velikost závěrného diodového svodového proudu lze vyjádřit jako $I_L = A_D J_S$ kde A_D je plocha difuze drainů a J_S je hustota svodového proudu. Tato hustota je určována technologií a je slabě závislá na velikosti napájecího napětí. Pro typický CMOS proces je tento proud silně závislý na teplotě (zdvojnásobení na zhruba každých $9^{\circ}C$). Pro čip s 1 milionem tranzistorů s průměrnou plochou drainů $10\,\mu m^2$ se celkový závěrný diodový svodový proud proud pohybuje řádově kolem $25\,\mu A$ [1]. Druhou komponentou svodových proudů je podprahový svodový proud, který je zapříčiněn difuzí mezi sourcem a drainem, když se tranzistor nachází ve slabě inverzním módu a pro napětí V_{GS} platí : $0 < V_{GS} < V_t$ kde V_t je prahové napětí. Velikost podprahového svodového proudu je exponenciálně závislá na teplotě a je dominantním faktorem statické spotřeby. Na obrázku 3 jsou potom ukázány obě složky vodových proudů. Červenou šipkou je na parazitní diodě D1 tranzistoru Q2 vyznačen závěrný diodový svodový proud a fialovou šipkou je potom na tranzistoru Q2 vyobrazen podprahový svodový proud v invertoru CMOS [1].



Obrázek 3: Základní složky svodových proudů

3.2 Optimalizace na technologické úrovni:

Jak je patrné ze vzorce 2 je dynamická spotřeba přímo závislá na V_{dd}^2. Jako jedna z možností snížení spotřeby se tedy nabízí právě snížení napájecího napětí CMOS obvodů, nicméně při snižování napájení obvodů se také významně snižuje průchodnost obvodu, jak ukazuje obr. 4 a vyjadřuje vzorec 4, kde V_{dd} je napájecí napětí, L je délka a W šířka kanálu tranzistoru, V_t je prahové napětí a C_L je kapacita hradla a C_{ox} je kapacita hradlového oxidu.

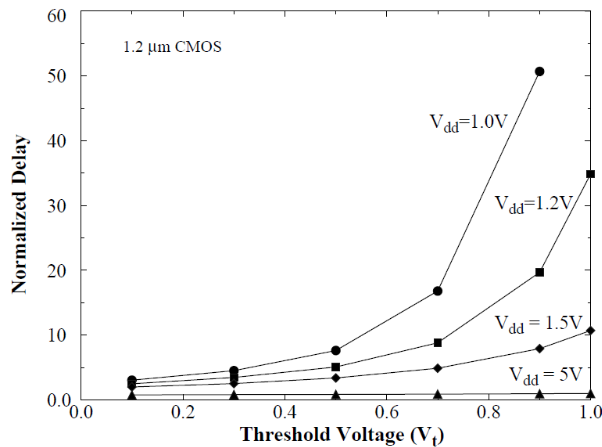


Obrázek 4: Zpoždění hradla v závislosti na napájecím napětí. Převzato z [1].

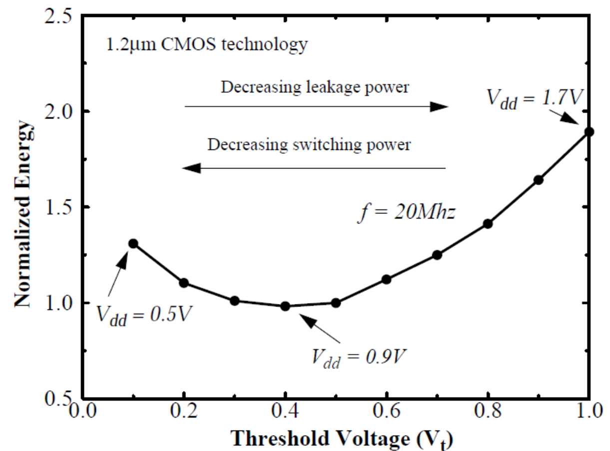
$$T_d = \frac{C_L \times V_{dd}}{I} = \frac{C_L \times V_{dd}}{\mu C_{ox}(W/L)(V_{dd} - V_t)^2} \quad (4)$$

Pro zachování průchodnosti dat CMOS obvodem je kromě použití paralelismu a pipeline (které budou probrány v kap. 3.4.1) jednou z možností snížení prahových napětí tranzistorů (obr. 5). Nicméně ani toto prahové napětí nelze snižovat do nekonečna, se snižujícím se prahovým napětím roste vliv podprahových svodových proudů. Proto existuje určité optimum, jak ukazuje obr. 6. Další možností pro snížení spotřeby CMOS obvodů je změna poměru W/L. Při změně poměru délky ku šířce kanálu (W/L) však spotřeba závisí také na celkové kapacitě, kterou bude tranzistor spínat, jak

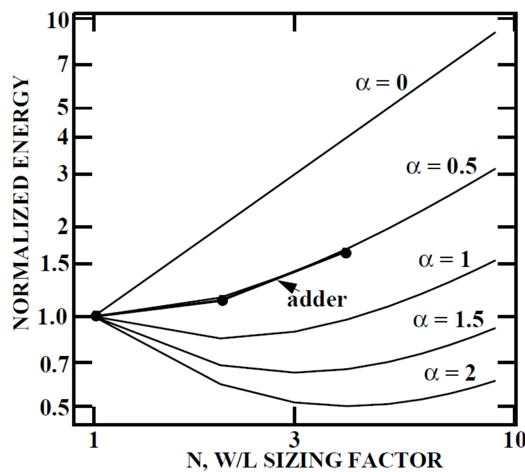
ukazuje obr. 7, kde α představuje poměr kapacity MOS tranzistorů ku parazitní kapacitě vedení. Jak je z obrázku 7 patrné pro větší α existuje optimum W/L a pro α=0 (nulová parazitní kapacita) se naopak se zvyšujícím se poměrem W/L spotřeba zvedá. Proto se pro vnitřní logiku, kde nedochází ke spínání tak velkých kapacit, používá jiná velikost tranzistorů než pro spínání větších kapacit mimo čip [1].



Obrázek 5: Efekt snižování prahového napětí na zpoždění při různých napájecích napětích. Převzato z [1].



Obrázek 6: Kompromis mezi dynamickými a leakage ztrátami. Převzato z [1].



Obrázek 7: Spotřeba energie ku poměru W/L pro různá α. Převzato z [1].

3.2.1 Minimalizace spínaných kapacit

Proměnnou C_L ve vzorci 1 je možné vyjádřit jako sumu následujících 3 kapacit : $C_L = C_{fo} + C_w + C_p$ kde

- C_{fo} je vstupní kapacita hradel na výstupu z daného hradla (*fan-out*)
- C_w je kapacita spoje (vedení)

- C_p je parazitní kapacita

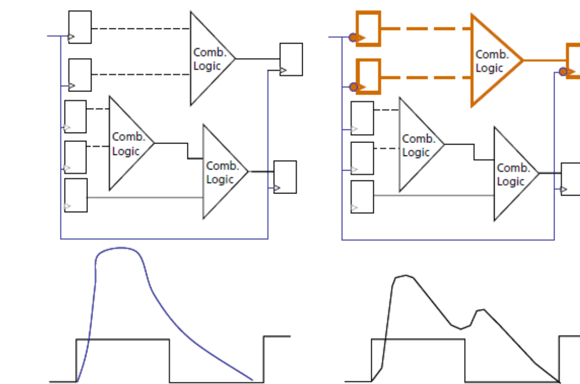
přičemž hodnota C_w je dominantní složka celkové kapacity C_L. Tuto kapacitu je možno ovlivnit návrhem layoutu celého čipu, kdy hradla, která mají mezi sebou největší spínací aktivitu, jsou umístěna blízko sebe a celkově je snaha vyhnout se dlouhým spojům přes celý čip [12].

3.3 Optimalizace na logické a RTL úrovni

Na RTL a logické úrovni je mnoho možností jak ovlivnit za pomoci změny spínací aktivity obvodu efektivní kapacitu $\alpha_{0\to 1} C_L$ a tím snížit spotřebu celého obvodu. Mezi techniky, které mají vliv na tuto změnu spínací aktivity, lze zařadit zejména hradlování hodinového signálu, redukce zákmitů, pořadí jednotlivých aritmetických operací a sdílení prostředků v obvodu. Kromě snižování spotřeby je také možno na této úrovni optimalizace spotřebu rovnoměrněji rozprostřít.

3.3.1 Snížení špičkové spotřeby

K řízení registrů je většinou použita jen jedna hrana hodinového signálu. To způsobuje náhlé zvýšení spotřeby obvodu během většinou vzestupné hrany hodinového signálu. Pro snížení této špičkové spotřeby je možno použít změny citlivosti jednotlivých registrů na typ náběžné hrany hodin. Toto opatření sice nesníží spotřebu, ale rovnoměrněji ji rozdělí (obr 8) [2].



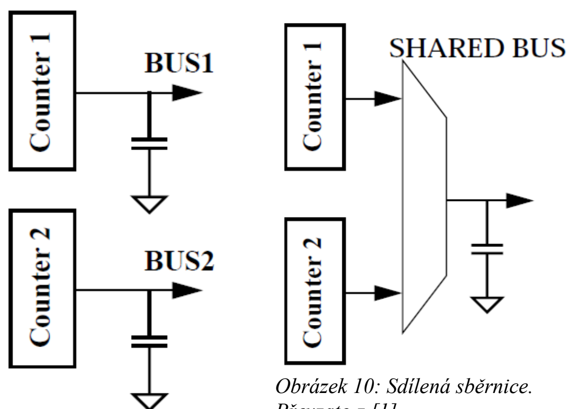
Obrázek 8: Rozprostření spotřeby. Převzato z [2].

3.3.2 Vliv sdíleni prostředků na spotřebu

Při návrhu může být snaha sdílet některé prvky a tím snížit plochu obvodu. Takovéto řešení však může vést ke zvýšení spínací aktivity a tím i ke zvýšení spotřeby.

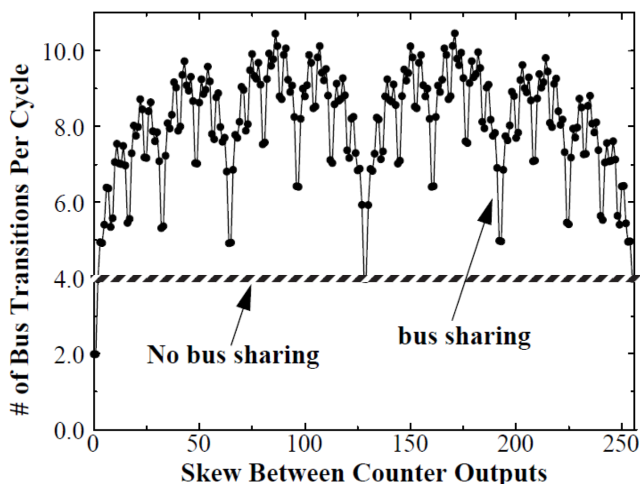
Sdílení sběrnic

V [1] je uvažován případ časového multiplexu sběrnice, kdy tato sběrnice běží na dvojnásobné frekvenci než 2 samostatné sběrnice (viz obr. 9 a 10). Na tyto sběrnice je zapojen výstup z čítače.



Obrázek 9: Nesdílená sběrnice. Převzato z [1].

Obrázek 10: Sdílená sběrnice. Převzato z [1].

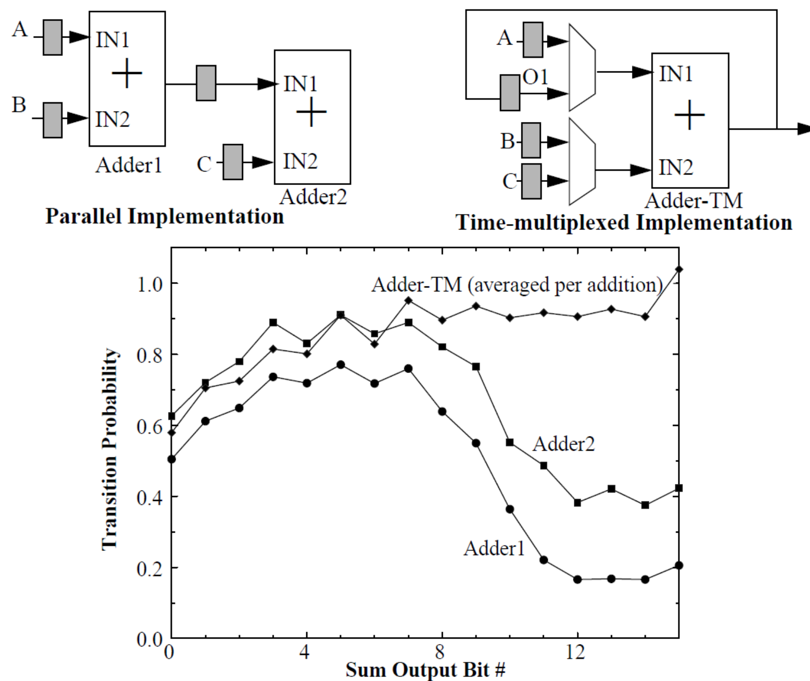


Obrázek 11: Porovnání spínací aktivity pro sdílenou a nesdílenou sběrnici pro čítač. Převzato z [1].

Pro oba případy byla spočítána pravděpodobnost spínací aktivity. Jak je patrné z obrázku 11, tak pouze pro nulový rozdíl mezi výstupem z čítačů (*skew*) je celková pravděpodobnost spínací aktivity sdílené sběrnice menší. Pro ostatní hodnoty je pravděpodobnost spínací aktivity větší (a to až více než dvojnásobně). Přestože tedy sdílená sběrnice spíná menší kapacitu než nesdílená, tak její spotřeba bude větší [1].

Sdílení aritmetických prvků

V [1] je opět uvažován jednoduchý případ filtru FIR druhého řádu a dva možné způsoby jeho implementace, jeden se dvěma paralelními sčítačkami a druhý s jednou časově multiplexovanou sčítačkou. Vstupní signál odpovídal běžné řeči (angličtina). Na těchto datech byla v simulátoru změřena spínací aktivita jednotlivých sčítaček. Na obrázku 12 je v grafu vynesena průměrná pravděpodobnost přechodu pro různě velké výstupy dat. Jak je z grafu patrné, ukázalo se, že průměrná pravděpodobnost přechodu je opět větší pro časově multiplexovanou sčítačku, zejména pro větší počet bitů [1]. Mohlo by se tedy zdát, že je vždy výhodnější vyhnout se sdílení aritmetických operátorů. V některých případech, kdy data vstupující do operátorů jsou stejná, je naopak sdílení žádoucí, jak je naznačeno v následujících ukázkách kódu ve VHDL v tabulce 1 [12].



Obrázek 12: Pravděpodobnost přechodu pro paralelní a časově multiplexovanou sčítačku. Převzato z [1].

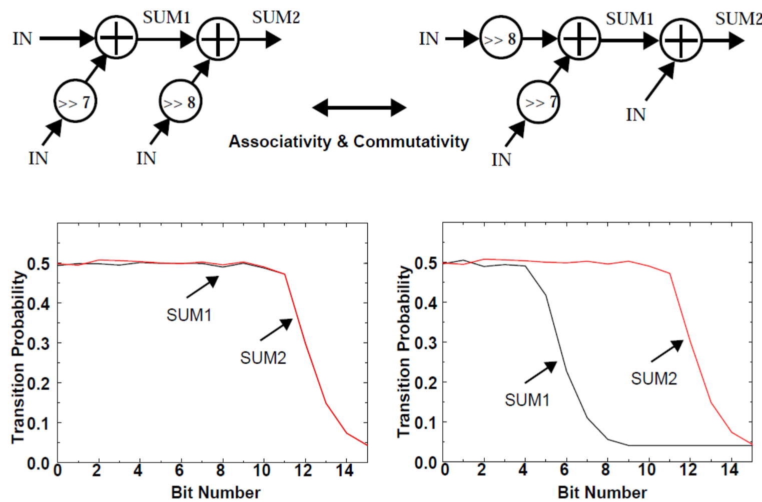
<table>
<tr><th>Nesdílení logiky</th><th>Sdílení logiky</th></tr>
<tr><td>mux: process (sel,value1,value2)
begin
case sel is
when "000" =>
output <= true;
when "001" =>
output <= false;
when "010" =>
output <= (value1 = value2);
when "011" =>
output <= (value1 /= value2);
when "100" =>
output <= (value1 < value2);
when "101" =>
output <= (value1 <= value2);
when "110" =>
output <= (value1 > value2);
when others =>
output <= (value1 >= value2);
end case;
end process mux;</td><td>cmp_equal <= (value1 = value2);
cmp_greater <= (value1 > value2);

mux: process (sel,cmp_equal,cmp_greater)
begin
case sel is
when "000" =>
output <= true;
when "001" =>
output <= false;
when "010" =>
output <= cmp_equal;
when "011" =>
output <= not cmp_equal;
when "100" =>
output <= not cmp_greater;
when "101" =>
output <= not cmp_greater or cmp_equal;
when "110" =>
output <= cmp_greater;
when others =>
output <= not cmp_greater or cmp_equal;
end case;
end process mux;</td></tr>
<tr><td>Výstup z ISE: 1 komparátor rovnost
2 komparátory <, ></td><td>Výstup z ISE: 1 komparátor rovnost
1 komparátor <, ></td></tr>
</table>

Tabulka 1: Sdílení prostředků. Převzato z [12].

3.3.3 Vliv pořadí operací na spotřebu

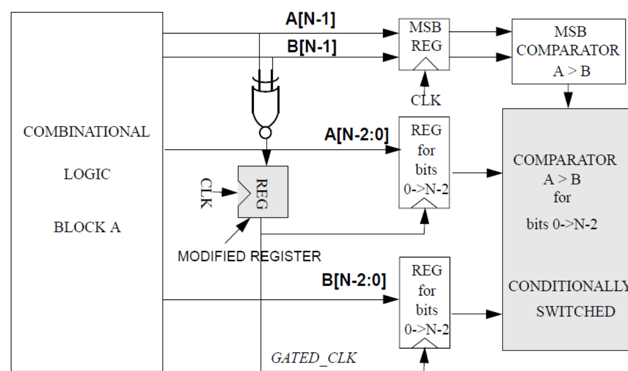
Pořadí operací v aritmetických operacích může mít také vliv na spotřebu. Různé pořadí operací (v rámci asociativity a komutativity) může mít různé pravděpodobnosti přechodů jednotlivých bitu a tím i různou efektivní kapacitu C_{ef} . Různé pravděpodobnosti přechodů pro různé pořadí vstupních signálů jsou dobře patrné na obrázku 12.



Obrázek 13: Redukování aktivity reorganizací vstupních signálů. Převzato z [1].

3.3.4 Hradlování hodin a vypínání části obvodu

Tato technika významně přispívá ke snižování spínací aktivity za cenu přidání několika málo řídících obvodů. Hlavní myšlenka spočívá ve vypínání hodin do těch částí obvodu, které nejsou momentálně zapotřebí. Příklad upravení komparátoru podle této filozofie je na obrázku 14. Zatímco u klasického komparátoru se porovnávají vždy celá slova, zde dojde k porovnání nejvýznamnějších bitů u obou slov. Pokud jsou schodné, je aktivován přes modifikační registr komparátor na zbytek slov[1].



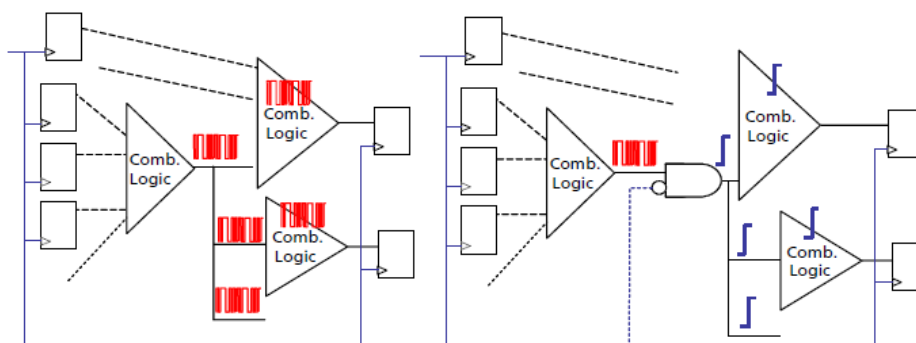
Obrázek 14: Ukázka hradlování hodin pro snížení spotřeby. Převzato z [1].

3.3.5 Redukce hazardů

Další technikou pro snížení spínací aktivity obvodu je redukce hazardů. Signál je považován za hazardní, pokud změní svoji hodnotu několikrát za periodu hodin a pouze poslední hodnota je využita. Jako zdroje hazardů můžeme uvést například nestabilní logické výrazy, nevyvážené délky cest, citlivé kombinační cesty nebo multiplexor, který se přepíná několikrát během jednoho hodinového cyklu. Hazardy lze rozdělit do dvou kategorií : hazardy v jednoduchých cestách a hazardy ve vícečetných cestách. Nástin odstranění hazardů ve vícečetných cestách lze popsat v následujících krocích:

1. Identifikace rychle spínacích vodičů a jejich řadičů
2. Odhad nejhoršího časového průběhu pro vstupy do řídících bloků
3. Posouvání zdroje hazardů nahoru nebo dolů v logice
4. Nový odhad nejhoršího časování pro vstupy řídících buněk
5. Vložení registru za řadič ovládaného opačnou hranou hodin v závislosti na předchozí časové analýze.

Odstranit hazardy v jednoduchých cestách lze například vložením fázově posunutých registrů, jejichž správné vložení může být velmi komplexní problém a může vést k velkým změnám chování obvodu. Na obrázku 15 je potom ukázána redukce hazardů u vícečetných cest[2].



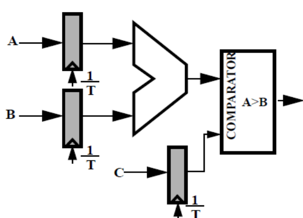
Obrázek 15: Redukce šíření hazardů. Převzato z [2].

3.4 Optimalizace na architektonické úrovni

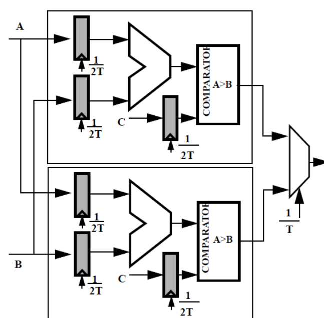
Na architektonické úrovni je opět možnost udělat několik kroků, které mohou významně přispět ke snížení spotřeby. Mezi tyto techniky patří například správná volba kódování, techniky pro zvýšení propustnosti dat obvodem, vhodný přístup k připojení pamětí a použití cache. [1, 2, 12].

3.4.1 Paralelismus a pipelining

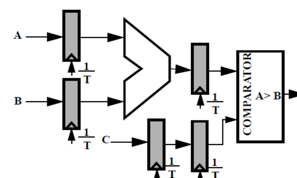
Další možností ke snížení spotřeby je podle vzorce 1 snížení napájení obvodu. Při snížení napájení obvodu se zvýší zpoždění na jednotlivých hradlech, a tím se zvětší zpoždění v kritických cestách v kombinační logice a dojde tedy ke snížení maximální frekvence, na které je schopen obvod pracovat. Existují dva přístupy jak tento pokles kompenzovat a zachovat tak propustnost obvodu. První možností je paralelismus. Při použití paralelismu se obvod rozdělí na několik částí, které danou část úlohy vykonají současně, tím je možno i při snížené pracovní frekvenci zachovat stejnou prostupnost dat obvodem(viz obr. 17). Druhou možností je použití pipeline. Pipeline rozdělí obvod za pomocí registrů na několik částí, a tím výrazně zkrátí nejdelší kombinační cestu, kterou musí signál urazit za hodinový cyklus. Tím dojde ke zvýšení maximální pracovní frekvence obvodu a je tak možno kompenzovat její pokles v důsledku sníženého napájecího napětí(viz obr. 18). Obě metody lze kombinovat [1]. V tabulce 2 je uvedeno, o kolik se sníží spotřeba a zvětší plocha obvodu při snížení napájení a zachování propustnosti dat obvodem.



Obrázek 17: Jednoduchá datová cesta. Převzato z [1].



Obrázek 16: Rozdělení datové cesty za pomoci paralelismu. Převzato z [1].



Obrázek 18: Rozdělení datové cesty za pomoci pipeline. Převzato z [1].

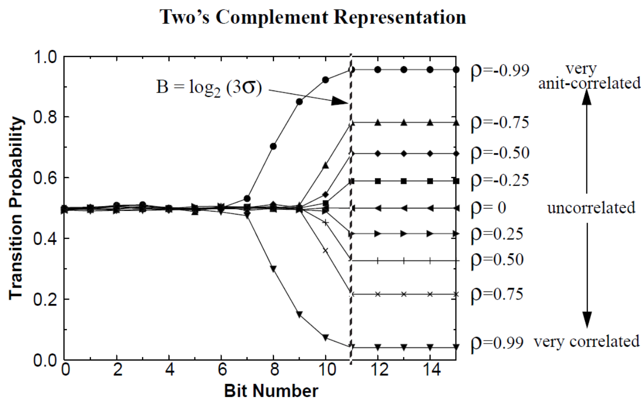
Architektura	Napětí	Plocha (normalizováno)	Spotřeba (normalizováno)
Jednoduchá	5,0 V	1	1
Paralelní	2,9 V	3,4	0,36
Pipeline	2,9 V	1,3	0,39
Pipeline-paralelní	2,0 V	3,7	0,2

Tabulka 2: Vliv architektury na snižování spotřeby. Převzato z [1].

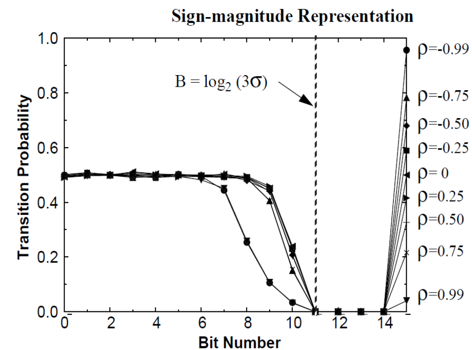
3.4.2 Volba číselné reprezentace

Dalším důležitým faktorem pro snížení spotřeby je volba vhodné soustavy pro číselnou reprezentaci. Vhodnou volbou číselné reprezentace (ale stejně tak i volbou kódování stavových automatů nebo čítačů) je možno snížit pravděpodobnost přechodu daného bitu a snížit tak efektivní kapacitu C_{ef} . Pro volbu vhodné číselné soustavy je nutné dobře promyslet jaké operace budeme

s čísly provádět (pro některé číselné reprezentace je nutno počítat se složitějšími sčítačkami popřípadě násobičkami) a také zvážit, jak hodně jsou mezi sebou jednotlivá data korelována. Ukázka pravděpodobnosti přechodu jednotlivých bitů pro různé formy korelace a pro různé číselné reprezentace je na obrázcích 19 a 20 [1, 2].



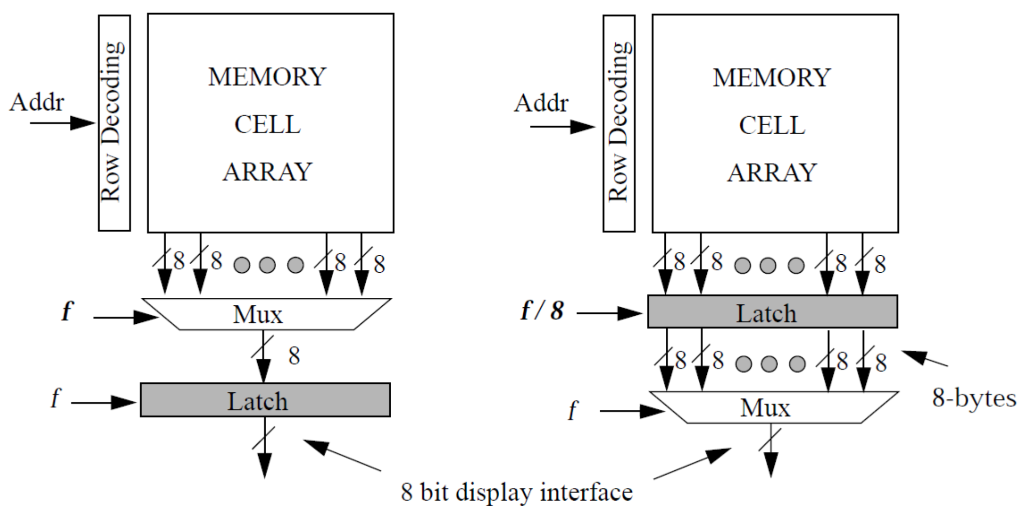
Obrázek 19: Pravděpodobnost přechodu jednotlivých bitů vůči stupni korelace dat - dvojkový doplněk. Převzato z [1].



Obrázek 20: Pravděpodobnost přechodu jednotlivých bitů vůči stupni korelace dat – znaménkové rozšíření. Převzato z [1].

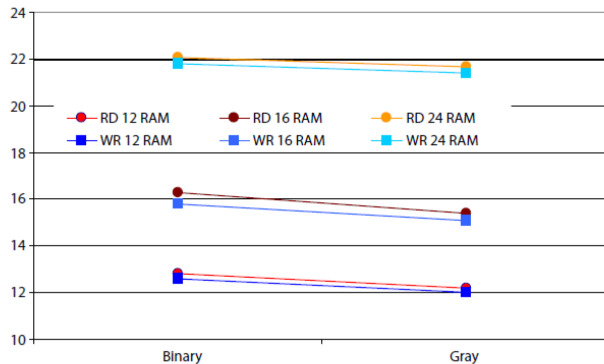
3.4.3 Přístup k pamětem

Paralelismus může být použit i pro přístup k pamětem. Při přístupu do paměti jsou dekódovány řádky a následně je multiplexorem vybrán správný sloupec, načež jsou data pod frekvencí f uloženy do výstupního registru. Tento postup lze však obrátit, kdy je nejprve celý řádek dat uložen do registru na frekvenci f/n kde n je šířka řádku, z něhož jsou pak postupně multiplexorem na frekvenci f vybírána data.

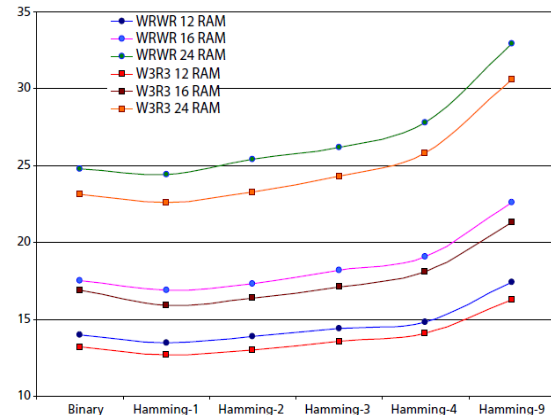


Obrázek 21: Rozdílné přístupy do paměti. Převzato z [1].

Díky tomuto opatření může být sníženo napájení paměti při stejné propustnosti dat a tím i ke snížení spotřeby, Celý přístup je vyobrazen na obrázku 21, kde vlevo je vyobrazen náhodný přístup do paměti a vpravo paralelní přístup. Takováto optimalizace za pomoci paralelního přístupu k datům je výhodná zejména tam, kde jsou data z principu sekvenční (video, audio apod...). Na spotřebu má vliv velikost pamětí a na spínací aktivitu, a tím i spotřebu, správná volba adresního kódování a také pořadí jednotlivých zápisů a čtení. Vliv má také adresní vzdálenost mezi čtenými nebo zapisovanými slovy, jak je ukázáno na následujících obrázcích 22 a 23 [1, 2].



Obrázek 22: Vliv čtení a zápisu na spotřebu při různých volbách adresního kódování a různé délce slova. Převzato z [2].



Obrázek 23: Vliv pořadí zápisů a čtení na spotřebu při různých adresních vzdálenostech a různé velikosti paměti. Převzato z [2].

3.4.4 Vliv jednotlivých opatření na spotřebu

V následujících tabulkách je zobrazen přehled jednotlivých technik pro snížení spotřeby a jejich přínosu k snížení spotřeby pro logickou, RTL a architekturní část. Jednotlivé techniky jsou rozděleny podle toho, ke které části obvodu patří (paměť, I/O, hodinový strom, logika). Poslední sloupeček určuje, zda-li je technika použitelná na FPGA či nikoliv.

Návrhové vylepšení	Předpoklady a alternativy	Odhad úspory(v závislosti na velikosti RAM)	FPGA
Kaskádní zapojení	Viz 3.4.3	2% - 5%	ANO
Hradlování hodinového signálu signály WE a RE	Hradlování hodinového signálu v nadřazené struktuře	5% - 10%	NE
Sekvence čtení a zápisu	Po několika čtecích operacích několik zápisů	2% - 5%	ANO
Úspěšné změny adresy	Zmenšení Hammingovy vzdálenosti	1% - 4%	ANO

Tabulka 3: Optimalizace spotřeby pamětí. Převzato z [2].

Návrhové vylepšení	Předpoklady a alternativy	Odhad úspory	FPGA
Zmenšení počtu vývodů I/O	Časový multiplex	Závisí na počtu I/O	ANO
Použití třístavové logiky na výstup místo klasické dvouúrovňové logiky pro výstup	Chytré zavedení signálů pro řízení třístavové logiky	Úměrně k povolení překlápění	ANO
Snížení I/O překlápěcí rychlost	Lepší kódování sběrnic	Závisí na kvalitě kódování sběrnic	ANO

Tabulka 4: Optimalizace spotřeby I/O. Převzato z [2]

Návrhové vylepšení	Předpoklady a alternativy	Odhad úspory	FPGA
Hradlování hodin	Vyžadována statická časová analýza a funkční validace	Může být velká ale závisí na designu	NE
Zavedení skupin registrů s opačnou citlivostí na hranu hodin	Časování musí dovolovat změnu	Redukce špičkového výkonu závisí na routování v logice ovládané redukovanými registry	ANO
Zavedení pipeline	Pokud jsou mezi stupni zpětné vazby vyžaduje detailní statickou časovou analýzu	10%+	ANO
Odstranění nadbytečných stupňů v pipeline	Vyžaduje funkční a časovou validaci	Závisí na počtu eliminovaných registrů	ANO
Výkonově řízený *place and route*	Nutnost kombinací s časovými podmínkami a časově řízeným place and route	Až k 10%	NE
Položení hodinového stromu	Analýza zahrnutých podmínek a potenciálních přetížení	Závisí na obsáhlosti hodinového stromu	NE

Tabulka 5: Optimalizace spotřeby hodinového stromu. Převzato z [2]

Návrhové vylepšení	Předpoklady a alternativy	Odhad úspory	FPGA
Plošně orientovaná syntéza	Statická časová analýza ukáže mezery	Může být velká ale závisí na designu	ANO
Aritmetické bloky s nízkou spotřebou	Časové a plošné atributy bloků vyžadovány	5% - 15%	ANO
Úsporné čítače	Pro obecné čítání je nejvhodnější binární, pro řízení sběrnic Grayův, pokus má být obsah čítače nějak interpretován nejvhodnější je kruhový čítač	Závisí na velikosti a počtu čítačů	ANO
RTL změny pro redukci glitchů	Vložení registrů do vícečetných cest a falešných cest	Závisí na velikosti logiky ve falešných a vícečetných cestách	ANO
	Vložení hradla AND řízeného opačnou hranou hodin	Závisí na velikosti logiky a aktivity glitchů	
	Vložení latchů řízených opačnou hranou hodin vyžaduje statickou časovou analýzu	Závisí na velikosti logiky a aktivity glitchů	
Výkonově řízený *place and route*	Nutná kombinace s časovými podmínkami a časově řízeným *place and route*	5% - 10%	NE

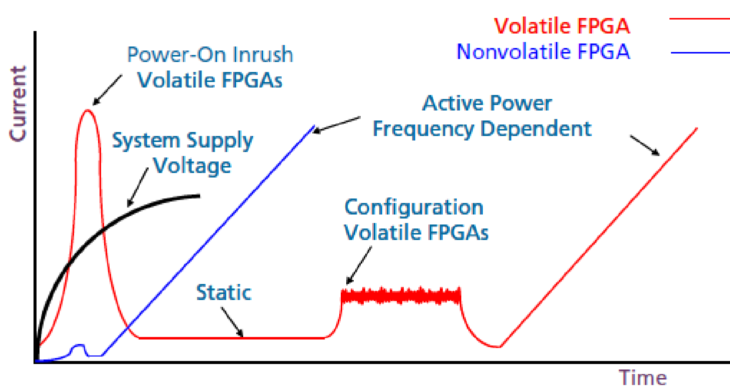
Tabulka 6: Optimalizace spotřeby logiky. Převzato z [2].

3.5 Optimalizace na systémové úrovni:

Při optimalizaci spotřeby na systémové úrovni je nutno provést několik úvah, které mohou významně přispět ke snížení spotřeby celého obvodu. Mezi tyto úvahy patří zejména výběr architektury, volba algoritmu, dekompozice mezi HW a SW. Na systémové úrovni je nutné zohlednit dynamický profil spotřeby, kdy je jasně vidět, které části obvodu mají největší podíl na spotřebě (I/O, paměti, kombinační logika, sekvenční logika), a právě na tyto části je nutné se zaměřit a provést optimalizaci [1, 2].

3.5.1 Výběr architektury

Při výběru vhodné architektury ponechejme stranou volbu mezi ASIC (*Application-specific integrated circuit*) a FPGA, která se odvíjí zejména od velikosti série, a zaměřme se na volbu volatilní/nevolatilní FPGA. Volatilní FPGA má dvě dodatečné složky spotřeby oproti nevolatilnímu FPGA, a to zvýšenou spotřebu během konfigurace a návalovou spotřebu během zapnutí, jak je ukázáno na obrázku 24. Volatilní FPGA má také větší statickou spotřebu a má i potřebu externí paměti PROM (*Programmable Read Only Memory*), ve které je uložena konfigurace [2]. I když je tedy nevolatilní FPGA výhodnější z hlediska spotřeby, volatilní FPGA jsou vyráběna pokročilejší technologií a jsou snáze dostupná. Z toho důvodu skupina pracující na systému BCI používá kit Digilent ATLYS s FPGA Spartan 6 fy. Xilinx.



Obrázek 24: Porovnání spotřeby u volatilních a nevolatilních FPGA. Převzato z [2].

3.5.2 Výběr algoritmu, dekompozice mezi HW a SW

Výběr správného algoritmu má velký vliv na snížení spotřeby. Při výběru algoritmu je třeba dbát na minimalizaci počtu vykonaných operací (a tím i snížení spínací aktivity) a také na možnost paralelizace a možnost použití pipeline. Dobrá paralelizovatelnost algoritmu může nejenom urychlit výpočet, ale díky možnosti snížit v takovém případě napájecí napětí může vést i ke snížení spotřeby [1]. Dalším důležitým rozhodnutím prováděným na systémové úrovni je dekompozice algoritmu mezi HW a SW. Na HW úrovni je dobré implementovat kritické části (smyčky, ve kterých algoritmus tráví nejvíce času), zbytek může být ponechán na SW úrovni. V [12] je uváděn případ, kdy v algoritmu HDTV Chromakey obsahujícím cca 22 tisíc řádek bylo pouze 15 řádek kódu (které se nejčastěji opakovali ve smyčkách) implementováno na HW úrovni a tím bylo dosaženo úspory energie 77%. Při návrhu SW je nutno věnovat také pozornost snižování spotřeby. Jedná se hlavně o slučování smyček, které jdou v programu za sebou (v každé smyčce je nutno inicializovat řídící proměnné a provádět jejich kontrolu), a také k vyvarování se příliš komplexních a složitých konstrukcí [12].

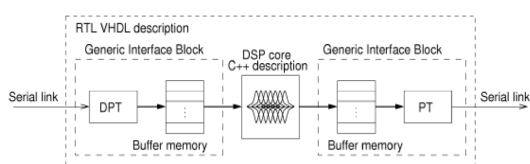
3.6 Shrnutí technik pro snížení spotřeby

Snižování spotřeby může být prováděno na všech úrovních návrhu, nicméně platí, že se vzrůstající úrovní abstrakce stoupá účinnost snižování spotřeby energie. Největší příležitosti ke snižování spotřeby jsou na systémové úrovni, zde lze snížit spotřebu řádově 10x – 100x [12]. Optimalizací na dalších úrovních návrhu lze spotřebu snížit řádově o desítky procent. V rámci vývoje systému BCI je používána deska ATLYS s FPGA SPARTAN 6. Zde samozřejmě není možno používat všechny úrovně návrhu snižování spotřeby. Není možno pracovat na technologické úrovni (změna prahových napětí a velikostí tranzistorů). Není možno snižovat napájecí napětí. Není možno optimalizovat hodinový strom a hradlovat hodinový signál. O to větší úsilí může být věnováno dalším úrovním snižování spotřeby systémovou úrovní počínaje a logickou a RTL konče. V tabulkách 3 - 6 jsou uvedeny některé techniky použitelné pro snižování spotřeby na FPGA na architektonické a logické a RTL úrovni. Je třeba také si uvědomit, že nejrychlejší řešení nejsou obvykle ta s nejnižší spotřebou. Při systémovém návrhu BCI je potřeba se zaměřit zejména na :

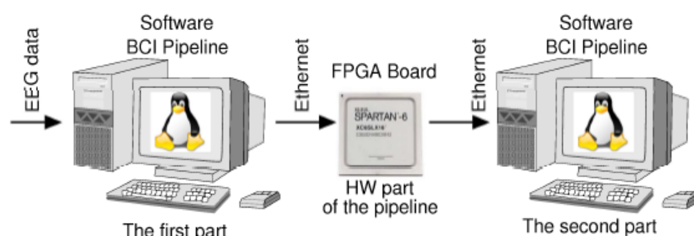
- správnou volbu číselného kódování
- správné sdílení zdrojů
- zvážit použití pipeline a paralelismu
- vhodné přistupování k pamětem
- minimalizaci spínací aktivity například minimalizací přenášených dat v rámci BCI

4 Systémový návrh hardwarové platformy BCI

Cílem je navrhnout hardwarovou akceleraci, která bude zachovávat modularitu používanou při dosavadním návrhu BCI. V článku [3] bylo představeno jedno takové řešení, kdy by se samotná výpočetní jádra DSP, která by zajišťovala filtraci, trigger a klasifikaci, byla navržena v jazyce C (za pomoci HLL syntézního nástroje). Tato výpočetní jádra DSP by byla „obalena“ bloky GIB (*General Interface Block*), které budou zajišťovat komunikaci mezi jednotlivými výpočetními moduly za pomoci sériového rozhraní (viz obrázek 26) a budou připojeny k hardwaru navrženém v diplomové práci [5] zajišťujícím komunikaci na rozhraní Ethernet. Na obrázku 25 je potom naznačeno propojení HW a SW části. Část, která bude implementována na FPGA, je vyznačena červeně na obrázku 1.



Obrázek 26: Schéma bloku GIB. Převzato z [3]



Obrázek 25: Spojení HW a SW části. Převzato z [3]

4.1 Typická práce systému

BCI je složitý systém. Jeho typickou práci lze popsat v několika bodech:

1. Konfigurace systému - po zapnutí je BCI nutné nastavit. Toto nastavení se děje za pomoci modulu *Control.* Systém BCI podporuje základní příkazy Load, Save a Reset. Tyto příkazy se zadávají pomocí příznaků zadaných do RTP paketů. Příkaz Load způsobí, že všechny bloky nahrají svoji konfiguraci ze souboru (tato konfigurace může být poměrně rozsáhlá – jedná se například o koeficinety u FIR filtru – cca. 200 údajů).

2. Trénování na konkrétní uživatele – do systému jsou vpuštěna trénovací data pro konkrétního uživatele. Tato data jsou do systému posílána dokud není vyhodnoceno, že jsou jednotlivé moduly dostatečně natrénované.

3. Normální provoz s uživatelem - systém BCI zpracovává data od uživatele. BCI má v závislosti na konfiguraci 2 - 45 elektrod o maximální vzorkovací frekvenci 256 - 1024 Hz při 16 bitovém slově. Maximální velikost čistého (bez hlaviček v RTP paketech) datového toku je tedy 738 kbit/sec [9].

4. Konec sezení – na konci sezení je možné uložit konfiguraci všech bloků příkazem Save do souboru. Příkaz Reset uvede všechny moduly do výchozího stavu.

4.2 Volba architektury

Správná volba architektury je důležitým krokem závisejícím na mnoha parametrech. V úvahu je třeba vzít propustnost systému pro daný datový tok a náročnost jednotlivých výpočetních operací. Každá z možných architektur (ať už CPU/DSP (*Central processing unit / Digital signal processor*) nebo FPGA) má několik výhod a nevýhod. Shrnutí hlavních výhod a nevýhod uvádím v tabulce 7.

Vlastnost	CPU/DSP	FPGA
Větší prostupnost díky nižšímu sdílení zdrojů	- CPU/DSP sdílí mnoho zdrojů (paměti apod.) což v systému tvoří úzké hrdlo	+ Na FPGA má každá úloha svoje zdroje
Práce s plovoucí řádovou čárkou	+ DSP může mít na tyto operace specializované bloky	- FPGA potřebuje specializované knihovny, je potřeba mnoho logiky
Paralelizovatelnost	- Jen v omezené míře některé části algoritmu	+ U době paralelizovatelných algoritmů velmi naroste výkon
Rychlost vývoje	+ Díky použití vyšších programovacích jazyků rychlejší	- Složitější vývoj nutná důsledná verifikace
Použití komplexních sběrnic, vyžadujících dodatečný software (USB, Ethernet)	+ Na CPU/DSP výrazně jednodušší implementace, pokud procesor obsahuje příslušné rozhraní	- U FPGA nutno použít speciální IP blok
Spotřeba elektrické energie	+ Vypínání nepoužívaných částí, uspávání	+ u dobře vyladěného systému může být poměr MIPS/Watt lepší než u CPU

Tabulka 7: Srovnání základních vlastností CPU/DSP a FPGA

Jak je patrné z tabulky 7 výhoda FPGA stoupá hlavně s velikostí datového toku a se složitostí prováděných operací. Celkový datový tok je 738 kb/s. Datový tok není zanedbatelný, navíc některé pokročilejší metody zpracování signálu, jejichž použití se pro BCI plánuje (např. skryté Markovské modely, Viterbiho dekodér), jsou velmi výpočetně náročné. Bloková struktura BCI podporuje paralelizaci, která je pro procesor obtížněji realizovatelná. Naměřená data jsou navíc posílána po síti Ethernet, jejíž obsluhou by byl procesor zatížen. Proto bylo vybráno řešení, kdy celé zpracování dat bude prováděno na jediném FPGA čipu. FPGA může mít díky paralelismu mnohem lepší poměr MIPS/Watt než procesor. Na tomto čipu bude umístěn soft procesor, který se bude starat o příjem a vysílání RTP paketů. Dále k tomuto procesoru budou připojeny dedikované výpočetní bloky, které

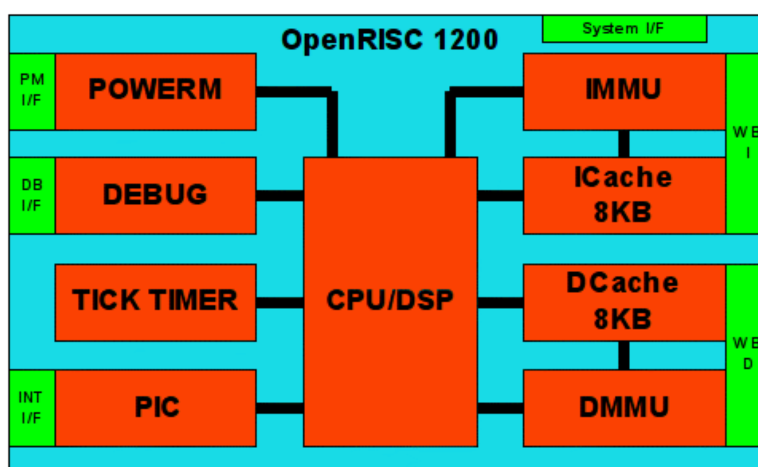
budou proudově zpracovávat data. Hlavní výhoda takového řešení spočívá v tom, že jednotlivé bloky bude snadné zaměňovat a navíc bude snadnější tyto bloky odladit.

4.3 Integrace s existujícím HW a SW

Všechny výpočetní bloky, které budou navrženy, budou muset být integrovány do již existujícího HW a SW [5]. Ten sestává z procesoru OpenRISC OR1200, na kterém běží operační systém FreeRTOS. Celý systém je schopen přijímat a odesílat RTP pakety po síti Ethernet. Tento procesor je vybaven sběrnicí Wishbone, na kterou je možno připojovat externí periferie. Právě na sběrnici Wishbone budou zapojeny výpočetní bloky (bloky GIB + DSP jádro). Základní schéma procesoru je na obr 27. Základní vlastnosti procesoru OpenRISC jsou potom [7] :

- open-source 32 bitový skalární RISC (*Reduced Instruction Set Computing*) procesor s Harvardskou architekturou
- pětistupňové vnitřní proudové zpracování dat (*pipeline*)
- přímo mapovaná instrukční a datová cache
- jednotka MAC a jednotka pro výpočty s plovoucí desetinou řádovou čárkou
- dvě rozhraní sběrnice Wishbone typu master, jedno rozhraní k datové a druhé připojené k instrukční sběrnici
- podpora GNU nástroji (kompiler, debuger)

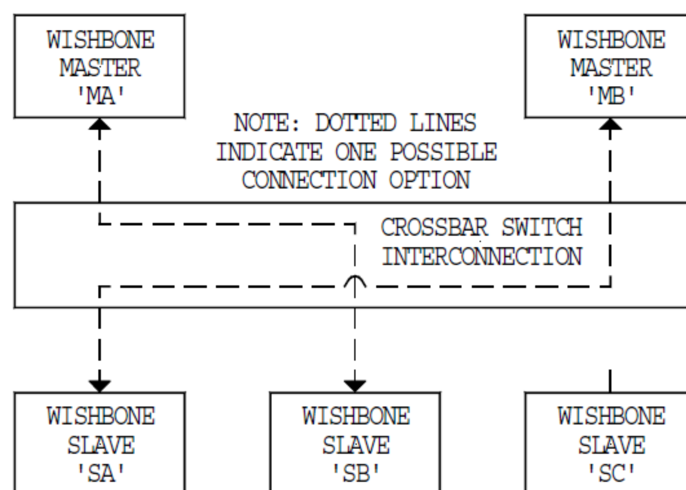
Protože na tuto sběrnici (Wishbone) budou připojeny výpočetní bloky, budu se jí věnovat detailněji.



Obrázek 27: Architektura OpenRISC OR1200. Převzato z [7].

4.3.1 Sběrnice Wishbone

Sběrnice Wishbone je určena a optimalizována pro použití v obvodech ASIC nebo FPGA. Jedná se o open-source sběrnice typu Master – Slave, skrze kterou je možno k procesoru OR1200 připojovat externí periferie. Jak bylo již zmíněno výše, procesor OR1200 má dvě sběrnice Wishbone. Jednu pro datovou a druhou pro instrukční sběrnici. Instrukční sběrnice provádí pouze čtení, zápis není podporován. Wishbone má šířku sběrnice 8 – 64 bitů, umožňuje připojení zařízení s menším datovým slovem, než je její nominální šířka, a signalizaci platných dat. Sběrnice Wishbone je korespondenční sběrnice typu Master – Slave. Master je zařízení, které může přistupovat ke sběrnici a ovládat zařízení typu Slave. Zařízení nejsou obvykle připojena přímo na sběrnici Wishbone, ale jsou připojena na arbitr sběrnice, který provádí dekódování adresy a propojuje jednotlivá zařízení. Při použití arbitru sběrnice se tak všechna zařízení, která jsou na sběrnici připojena, zdají být v jednotném adresním prostoru. Arbitrů sběrnice je podle [11] definováno více druhů. Jako nejlepší se však jeví arbitr typu c*rossbar switch*, který umožňuje paralelní propojení více zařízení. Schéma takovéhoto arbitru je na obrázku 28. Detailnější popis sběrnice Wishbone je potom obsažen v [5] a [11]



Obrázek 28: Arbitr typu Crossbar switch. Převzato z [11]

Signály na sběrnici Wishbone

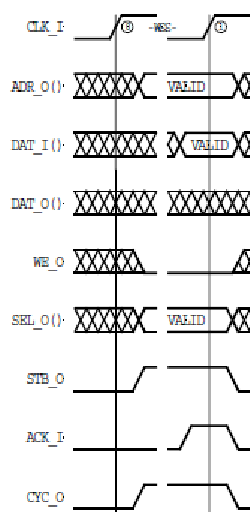
Sběrnice Wishbone má několik povinných a několik volitelných signálů. Signály, které jsou použity v návrhu BCI, jsou obsaženy v tabulce 8.

signál	šířka	popis	směr	
			master	slave
clk	1	hodiny	i	i
rst	1	reset	i	i
dat_i	32	vstupní data	i	i
dat_o	32	výstupní data	o	o
adr	32	adresa	o	i
sel	4	indikuje kde se nacházejí validní data	o	i
cyc	1	validní cyklus sběrnice	o	i
stb	1	začátek rámce	o	i
we	1	čtení/zápis	o	i
ack	1	indikuje normální ukončení cyklu sběrnice	i	o
err	1	indikace chyby	i	o
rty	1	opakování cyklu	i	o

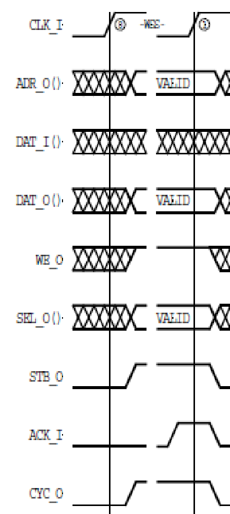
Tabulka 8: Signály na sběrnici Wishbone

Čtení a zápis dat na sběrnici Wishbone

Typické průběhy čtení a zápisu jsou na obrázcích 29 a 30.



Obrázek 29: Typický průběh čtení na sběrnici Wishone.Převzato z [11]



Obrázek 30: Typický průběh zápisu na sběrnici Wishbone.Převzato z [11]

Typický průběh čtení a zápisu

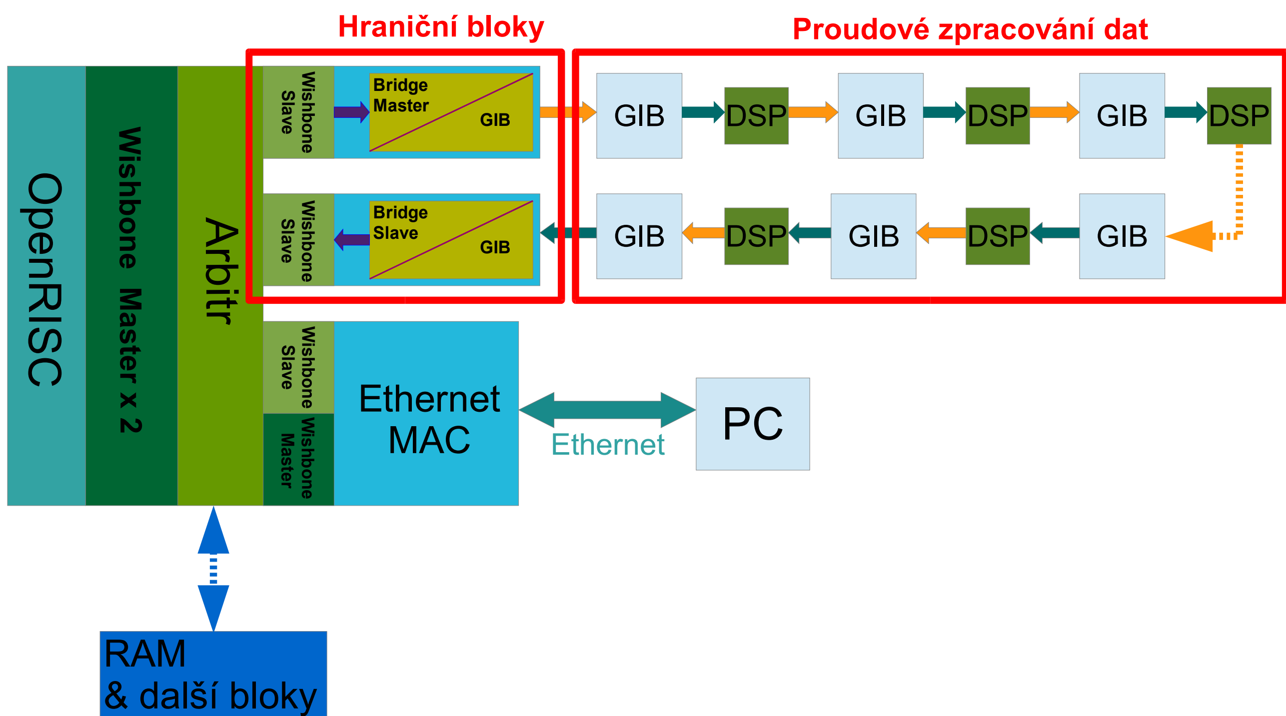
1. Master nastaví validní adresu, nastaví signál *sel* podle toho, kde jsou očekávána validní data, nastaví signály *cyc* a *stb* do log. 1. Pokud se jedná o zápis, nastaví do log.1 i signál *we* a nastaví validní data.

2. Při následující vzestupné hraně hodin Slave dekóduje příslušnou adresu. Pokud se jedná o čtení, vystaví příslušná data. Pokud o zápis, uloží je. Jako potvrzení správnosti dat nastaví signál *ack* do log. 1.

3. Master monitoruje signál *ack.* Pokud se jedná o čtení dat tak při vzestupné hraně hodin pokud je signál *ack* v log. 1, tato data uloží. Poté nastaví signály *cyc* a *stb* do log. 0. Po té co jsou signály *cyc* a *stb* v log. 0, Slave nastaví signál *ack* do log. 0.

4.4 Proudové zpracování dat

Proudové zpracování dat bude vytvořeno z DSP bloků, které bude za pomoci bloků GIB možné řadit za sebe. V původní koncepci v článku [3] bylo představeno, že by jednotlivé DSP bloky byli obaleny bloky GIB-Master a GIB-Slave, které by si přeposílali data. Rozhodl jsem se však, že by bylo lepší, kdyby se tyto bloky sloučili do jediného bloku GIB, který bude fungovat zároveň jako Master i Slave, a tím dojde k výraznému snížení spínací aktivity a tím i spotřeby. Pro N bloků DSP bude zapotřebí N+1 bloků GIB. Kromě toho bude nutno tuto výpočetní posloupnost připojit ke sběrnic Wishbone. K tomu účelu budou sloužit speciální hraniční bloky. Schéma zapojení celého HW BCI je na obrázku 31.



Obrázek 31: Schéma zapojení proudového zpracování dat

4.4.1 Bloky DSP

Bloky DSP budou provádět samotné zpracování EEG signálů. Tyto bloky budou psané v jazyce C za pomoci syntézního HLL nástroje nebo implementovány na úrovni RTL v jazyce VHDL. Bloky DSP budou zpracovávat EEG data, která budou načítat z pamětí RAM (*Random-access memory*), a zpracovaná data budou ukládat do další paměti RAM. Tyto paměti RAM budou uvnitř bloků GIB, které budou DSP bloky obalovat. Z jednoho bloku GIB bude DSP blok data načítat a do druhého ukládat výsledky. Blok DSP bude zpracovávat vždy jeden paket. Dále bude těmto blokům také potřeba posílat specifické nastavení (například koeficienty u FIR filtru).

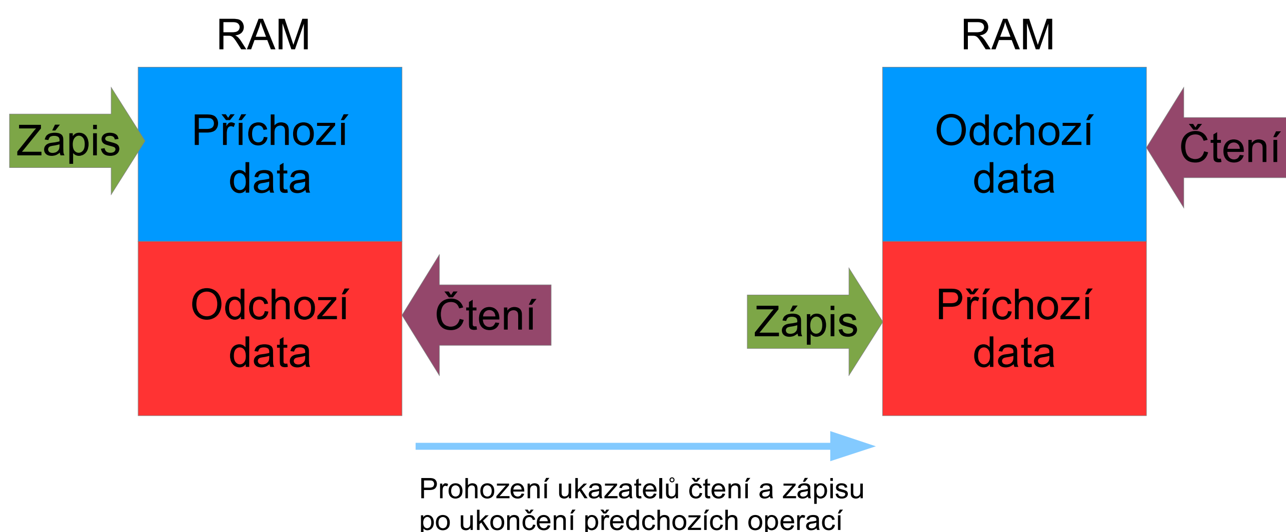
4.4.2 Bloky GIB

Bloky GIB budou tvořit prostředníka mezi jednotlivými výpočetními jádry DSP. Tyto bloky budou zajišťovat veškerý přísun dat pro bloky DSP. Bloky GIB budou zároveň přijímat i vysílat data. Každý blok GIB bude obsahovat dvouportovou paměť RAM, do které z jedné strany budou data ukládána a z druhé strany zároveň čtena. Zabraňovat kolizím dat bude interní logika. GIB bude navržen genericky, aby bylo snadné měnit velikost interní paměti. GIB bude muset implementovat několik rozhraní :

- Rozhraní pro konfiguraci – toto rozhraní se bude starat o nahrání konfigurace do jednotlivých bloků DSP.
- Rozhraní pro zápis do bloku GIB (příjem dat z DSP)
- Rozhraní pro čtení z bloku GIB (zápis do DSP)

Princip funkce

Hlavním úkolem, který GIB obstarává, je přeposílání jednotlivých EEG paketů dále tak, aby celý výpočetní řetězec pracoval jako pipeline. Aby se nemuselo čekat, než se zpracovaný paket překopíruje do dalšího bloku, pracuje GIB s dvěma pakety naráz. V jeden okamžik může být jeden paket do bloku GIB zapisován (ukládání zpracovaných dat) a druhý paket z něj čten (data, která jsou zpracovávána). Protože jsou oba pakety uloženy v jedné vnitřní paměti RAM, musí se interní logika starat o zabránění kolizí přepínáním jednotlivých adresních prostorů. Dá se tedy říci, že se GIB chová jako paměť typu FIFO (*First In, First Out)* o velikosti dvou položek (v paměti jsou uloženy dva pakety EEG dat). Pro větší přehlednost je vše zobrazeno na obrázku 32.



Obrázek 32: Princip funkce bloku GIB

Rozhraní pro konfiguraci

Jednotlivé DSP bloky je před použitím nutno nakonfigurovat (například nahrání koeficientů u FIR filtru). Tuto konfiguraci je možno provést buďto přes sběrnici Wishbone, která bude zavedena ke každému bloku GIB, nebo za pomoci specializovaných paketů, které by přenášely konfiguraci. Hlavní nevýhodou při použití nahrávání konfigurace přes sběrnici Wishbone by byla větší složitost bloků GIB (každý blok by musel implementovat rozhraní Wishbone). Navíc by bylo zapotřebí dlouhé sběrnice, tím pádem by docházelo ke spínání velkých kapacit a došlo by ke zvýšení spotřeby a snížení pracovní frekvence. Z tohoto důvodu bude konfigurace nahrávána prostřednictvím specializovaných paketů. Toto bude sice klást zvýšené nároky na bloky DSP, které budou muset tyto pakety interpretovat a nastavit se podle nich, ale celkové řešení bude jednodušší než při použití sběrnice Wishbone.

Rozhraní pro zápis do bloku GIB

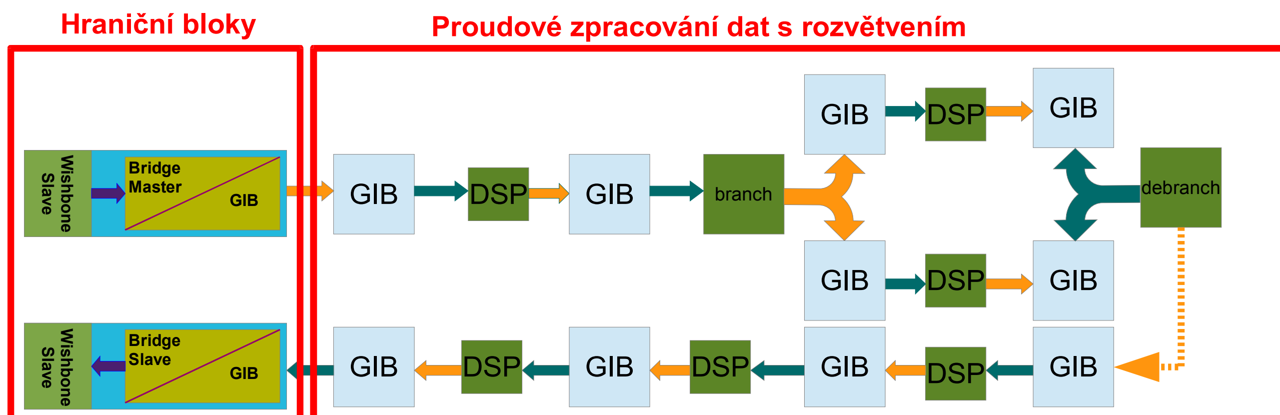
Blok DSP musí být schopen zapisovat data do paměti RAM umístěné v bloku GIB. DSP má k této paměti plný přístup. Blok GIB se stará o to, aby nedocházelo ke kolizím při čtení a zápisu v interní paměti bloku GIB. Informuje blok DSP o tom, že je paměť připravena pro zápis dat do interní paměti RAM. Blok DSP musí být schopen informovat blok GIB o tom, že s daty již nebude dále manipulováno a mají být zpřístupněny dalšímu bloku DSP ve výpočetní posloupnosti. Rozhraní je v obrázku 31 naznačeno oranžovou šipkou.

Rozhraní pro čtení z bloku GIB

Blok DSP musí být schopen číst data z paměti RAM umístěné v bloku GIB. DSP blok musí mít k této k této paměti plný přístup a moci libovolně číst. Kromě toho musí blok GIB informovat blok DSP o tom, že jsou k dispozici nová data, a blok DSP musí mít možnost informovat blok GIB o tom, že data již přečetl a nejsou dále potřeba . Toto rozhraní je vyznačeno na obrázku 31 modrou šipkou.

4.4.3 Další typy bloků v proudovém zpracování dat

Kvůli snadnější implementaci proudového zpracování dat by bylo výhodné výpočetní posloupnost v některých případech dále rozvětvit. Proudové zpracování dat by se doplnilo o rozvětvovací bloky, kdy by stejný druh výpočtu běžel nad různými daty (různé části RTP paketu), popřípadě různé druhy výpočtů (například výpočet horní a dolní propusti) nad stejnými daty. Jednotlivé větve se však před vstupem do mostu (hraničního bloku) musí opět složit. Rozvětvovací a slučovací bloky budou umístěny místo DSP bloku, jak je naznačeno na obrázku 33 a budou psané v jazyce C za pomoci syntézního HLL nástroje nebo implementovány na úrovni RTL v jazyce VHDL stejně jako bloky DSP.



Obrázek 33: Proudové zpracování dat s rozvětvením

4.5 Hraniční bloky

Přes hraniční bloky je celá výpočetní posloupnost připojena k procesoru OpenRISC 1200. Tyto bloky tedy představují jakýsi most mezi sběrnicí Wishbone a rozhraním, kterým disponují GIB bloky. Z jednoho hraničního bloku budou data vyčítána prvním GIB blokem a do druhého hraničního bloku budou zase data ukládána posledním GIB blokem ve výpočetní posloupnosti. Řídící signály pro blok GIB bude ovládat procesor prostřednictvím registrů, které bude obsahovat hraniční blok. Tyto bloky budou navrženy na úrovni RTL v jazyce VHDL.

4.6 Minimalizace spotřeby

Snížení spotřeby bude dosaženo co největším snížením spínací aktivity. Toho bude dosaženo jednak minimalizací spínací aktivity v bloku GIB a také může být spínací aktivita snížena odstraněním pro výpočet redundantních informací obsažených v RTP paketech. RTP paket má strukturu podle tabulky 9.

MAC header	IP header	UDP header	RTP header	EEG header	EEG data

Tabulka 9: Struktura RTP paketu. Převzato z [8]

Červeně jsou označeny informace, které nejsou pro výpočet potřeba a výpočetní DSP jádra nijak nebudou informaci v nich obsaženou během výpočtu měnit. Tyto hlavičky je možno při příjmu uložit, např do velkokapacitní DDR (*double-data-rate synchronous dynamic random access memory*) paměti připojené k FPGA a do dedikovaných výpočetních modulů poslat ořezané RTP pakety a po té, co projdou výpočetními moduly je opět složit a odeslat za pomoci Ethernetu ven z FPGA. Kromě těchto hlaviček nejsou potřeba ani všechny údaje obsažené v hlavičce EEG. Její detailní uspořádání je uvedeno v tabulce 10. V této tabulce jsou červeně údaje, které nejsou pro výpočet potřeba, fialově potom údaje, které se v průběhu výpočtu nemění a je tedy výhodné předat je jako generickou konstantu. Tyto údaje (červené a fialové) se tedy kvůli úspoře energie nebudou do bloku GIB posílat. Struktura paketu, který se bude posílat do bloků GIB, je uveden v tabulce 11. Všechny údaje v hlavičkách budou zarovnány do 4B kvůli snadnějšímu zpracování blokem DSP.

4.6.1 Odhad velikosti RAM v bloku GIB

Ze znalosti struktury paketu, který bude posílán do DSP jader, je možno provést horní odhad velikosti paketu přeposílaného mezi bloky GIB. Z toho lze odvodit maximální velikost paměti RAM obsaženou v GIB. Na základě [9] jsem provedl následující odhad:

$$EEG_{header}=20+4*NCHAN+aditional_{header}=20+4*45+100=300[B]$$

$$EEG_{data}=sampl_{num}*NCHAN*2=100*45*2=9000[B]$$

$$GIB_{\text{packet size}}=EEG_{header}+EEG_{data}=9300[B]$$

Kde $NCHAN$ je počet kanálů, $sampl_{num}$ je počet vzorků v jednom paketu, $aditional_{header}$ je dodatečná informace v paketu o maximální velikosti 100B. Data musí být násobeny 2, protože data jsou 16 bitová.

GIB z principu své funkce potřebuje dvojnásobnou velikost paměti RAM než je maximální velikost paketu v něm uložená. Celkem tedy 18600 B. Spartan 6 XC6SLX45, který je na vývojové desce

ALTYS má k dispozici 2088Kb paměti BRAM rozdělené po 116 18kb blocích. To znamená, že jeden blok GIB bude obsahovat až 5 bloků BRAM (bloky mezi jednotlivými bloky GIB nelze sdílet). Na jednom čipu Spartan 6 XC6SLX45 může být tedy až 23 bloků GIB.

Offset	Pole	Velikost pole	Datový typ	Popis
0	FRMT	4B	unsigned integer	Packet identification. STOP packet has the value of FRMT 0 and normal packets have the value of FRMT 2 or more.
4	BlockNO	4B	unsigned integer	Packet sequence number.
8	FVZ	2B	unsigned short integer	Sampling rate in Hz.
10	BitUV	2B	unsigned short integer	Sensitivity in Bit / uV
12	NCHAN	2B	unsigned short integer	Number of channels in packet.
4	NSMPL	2B	unsigned short integer	Number of samples per channel.
16	DOFFS	**2B**	unsigned short integer	Data offset in the packet
18	RES	**2B**	unsigned short integer	Reserved...
20-(DOFFS-1)	ADDH	**DOFFS-18**	array of signed short integer	Additional header.
DOFFS	ImpDATA	**2·NCHAN [B]**	array of signed short integer	Impedance of each EEG channel.
DOFFS+ 2·NCHAN [B]	DCDATA	**2·NCHAN [B]**	array of signed short integer	DC component of each EEG channel.
DOFFS+ 4·NCHAN [B]	DATA	**2 · NCHAN · NSMPL [B]**	array of signed short integer	Size=2· NCHAN · NSMPL [B]. Data are sent as they were picked-up. SMPL1 CH1, CH2, ... , CHx SMPL2 CH1, CH2, ... , CHx ... SMPLy CH1, CH2, ... , CHx SMPL ... sample, CH ... channel, x = NCHAN, y = NSMPL

Tabulka 10: Struktura hlavičky EEG a EEG dat v RTP paketu. Převzato z [8].

Offset	Pole	Velikost pole	Datový typ	Popis
0	FRMT	4B	unsigned integer	Packet identification. STOP packet has the value of FRMT 0 and normal packets have the value of FRMT 2 or more.
4	BitUV	4B	unsigned short integer	Sensitivity in Bit / uV
8	NSMPL	4B	unsigned short integer	Number of samples per channel.
10	DOFFS	**4B**	unsigned short integer	Data offset in the packet
14-(DOFFS-1)	ADDH	**DOFFS-10**	array of signed short integer	Additional header.
DOFFS	ImpDATA	**2·NCHAN [B]**	array of signed short integer	Impedance of each EEG channel.
DOFFS+ 2·NCHAN [B]	DCDATA	**2·NCHAN [B]**	array of signed short integer	DC component of each EEG channel..
DOFFS+ 4·NCHAN [B]	DATA	**2 · NCHAN · NSMPL [B]**	array of signed short integer	Size=2· NCHAN · NSMPL [B]. Data are sent as they were picked-up. SMPL1 CH1, CH2, ... , CHx SMPL2 CH1, CH2, ... , CHx ... SMPLy CH1, CH2, ... , CHx SMPL ... sample, CH ... channel, x = NCHAN, y = NSMPL

Tabulka 11: Struktura dat posílaných do bloku GIB. Převzato z [8].

4.7 Koncept ovládacího SW

Ovládací SW bude nahraný v paměti připojené k procesoru OpenRISC. Chování tohoto programu se bude skládat z několika na sebe navazujících kroků, které se budou ve smyčce opakovat.

1. Příjem dat po rozhraní Ethernet – O příjem paketů se stará samostatný blok. Po přijetí paketu je tento paket rovnou uložen na příslušnou adresu do paměti. O přijetí paketu se procesor dozví buďto přerušením nebo čtením příslušného status registru. V této fázi se také procesor stará o neúplné nebo chybějící pakety (viz. kapitola 4.7.1).
2. Nahrání paketu přes hraniční blok do prvního bloku GIB - V této fázi budou příslušná data,

která jsou potřeba, nahrána do výpočetní posloupnosti (viz tabulky 9 a 11). Ostatní data mohou být ponechána v paměti nebo uloženy do DDR RAM. Procesor se také bude muset postarat o obsluhu řídících signálů bloku GIB prostřednictvím registrů umístěných v hraničním bloku. V této fázi by bylo výhodné použit DMA.

3. Vyzvednutí zpracovaného paketu přes hraniční blok z posledního bloku GIB – Po zpracování paketu výpočetní posloupností bude nutné vyzvednout data z hraničního bloku. O tom, že jsou data připravena k vyzvednutí, bude procesor informován prostřednictvím přerušení nebo čtením příslušného registru. Procesor překopíruje zpracovaný paket zpět do paměti. V této fázi by bylo výhodné použit DMA.
4. Odeslání zpracovaného paketu přes rozhraní Ethernet – V tomto kroku procesor složí zpět zpracovaný paket spolu s hlavičkami, které nebyly pro výpočet potřeba, a nastaví Ethernetový blok, aby odeslal data ven z FPGA.

4.7.1 Neúplné a chybějící pakety

Protože jsou RTP pakety přijímané po síti Ethernet, může dojít ke ztrátě paketu, jeho poškození nebo prohození pořadí jednotlivých paketů. Následující tabulka shrnuje události, které mohou nastat, a jejich řešení, které provede procesor OpenRISC.

Situace	Řešení
Paket je poškozen	Do výpočetní posloupnosti nejsou vpuštěny žádná data, paket je nahrazen umělým prázdným paketem.
Pakety jsou prohozeny	Je zpracován pouze první příchozí paket(novější paket), starší paket je zahozen a nahrazen prázdným paketem. Pakety ven jsou odeslány v pořadí prázdný paket, zpracovaný paket.
Několik paketů je vynecháno	Chybějící pakety jsou ignorovány, pokud přijdou chybějící pakety později jsou ignorovány a nahrazeny prázdnými pakety.
Pakety nechodí	Jsou generovány prázdné pakety v pravidelných intervalech.
Přijde více paketů naráz	Starší pakety jsou ignorovány, zpracován je jen nejnovější paket.

Tabulka 12: Řešení problému s RTP pakety

5 Návrh bloku GIB

Bloky GIB představují jakési spojovací články mezi jednotlivými jednotkami DSP. Každý blok GIB obsahuje paměť RAM, ze které jsou čtena data a do které je zapisováno. Kromě toho obsahuje interní logiku pro zabránění kolizí při čtení a zápisu. Protože na snížení spotřeby má výrazný vliv pracovní frekvence, GIB je navržen tak, aby každý z bloků DSP, který je k němu připojen mohl pracovat na jiné hodinové frekvenci. GIB je navržen genericky, aby bylo možné snadno měnit velikost vnitřní dvouportové paměti RAM. Dále je navržen tak, aby se co nejvíce snížila spínací aktivita paměti RAM.

5.1 Popis rozhraní

GIB implementuje rozhraní pro čtení a zápis z bloků GIB. Shrnutí a popis všech signálu je v následujících kapitolách.

5.1.1 Rozhraní pro zápis do bloku GIB

Toto rozhraní je na obrázku 31 vyznačeno oranžovou šipkou. Kromě standardních signálů pro zápis do paměti RAM v bloku GIB (adresa, data, povolení zápisu) obsahuje také dva korespondenční signály pro zabránění kolizí při zápisu do RAM. Signál *free_n* slouží pro informování bloku DSP o tom, jestli je v RAM místo na další paket. Pokud je signál *free_n* v log. 0 znamená to, že je možné do paměti jeden paket zapsat. Pokud chce blok DSP zapsat do GIB data a signál *free_n* je v log. 0, blok DSP nastaví signál *busy_in* do log.1, poté co se *free_n* překlopí do log. 1 může zapisovat data. Následně po dokončení zápisu je signál *busy_in* překlopen do log. 0. Pokud se signál *free_n* překlopí do log. 0, může se celá sekvence opakovat. Shrnutí všech signálu je v tabulce 13. Na obrázku 34 je vyobrazení korespondenčního režimu.

Obrázek 34: Korespondeční signály pro zápis do RAM

Signál	Směr	Typ	Popis
data_in	in	std_logic_vector	data
adr_in	in	std_logic_vector	adresa
we	in	std_logic	povolení zápisu
busy_in	in	std_logic	DSP signalizuje, že jsou do GIBu ukládána data
free_n	out	std_logic	GIB signalizuje , že je možné zapsat data

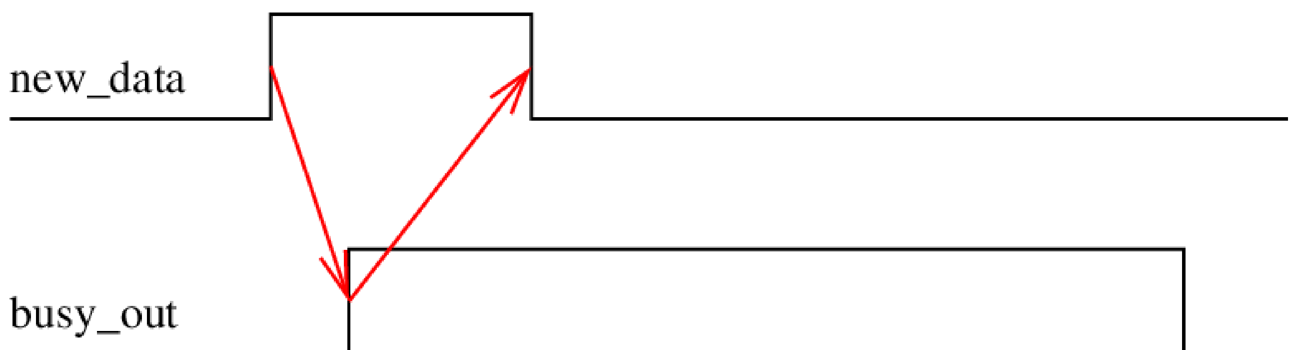
Tabulka 13: Rozhraní pro zápis do bloku GIB

5.1.2 Rozhraní pro čtení z bloku GIB

Toto rozhraní je na obrázku 31 vyznačeno modrou barvou. Kromě standardních signálů pro čtení z paměti RAM (adresa, data, povolení k čtení) obsahuje také dva korespondenční signály pro zabránění kolizí při čtení z RAM. Signál *new_data* v úrovni log. 1 signalizuje, že jsou k dispozici nová data. Pokud je chce blok DSP číst, musí nastavit signál *busy_out* do log. 1, poté, co se signál *new_data* přepne do log. 0, může číst data. Po přečtení dat blok DSP nastaví signál *busy_out* do log. 0 a celá sekvence se může opakovat.

Signál	Směr	Typ	Popis
data_out	out	std_logic_vector	data
adr_out	in	std_logic_vector	adresa
re	in	std_logic	povolení čtení
busy_out	in	std_logic	DSP signalizuje, že čte data z GIBu
new_data	out	std_logic	GIB signalizuje, že jsou k dispozici nová data

Tabulka 14: Rozhraní pro čtení z bloku GIB



Obrázek 35: Korespondenční signály pro čtení z RAM

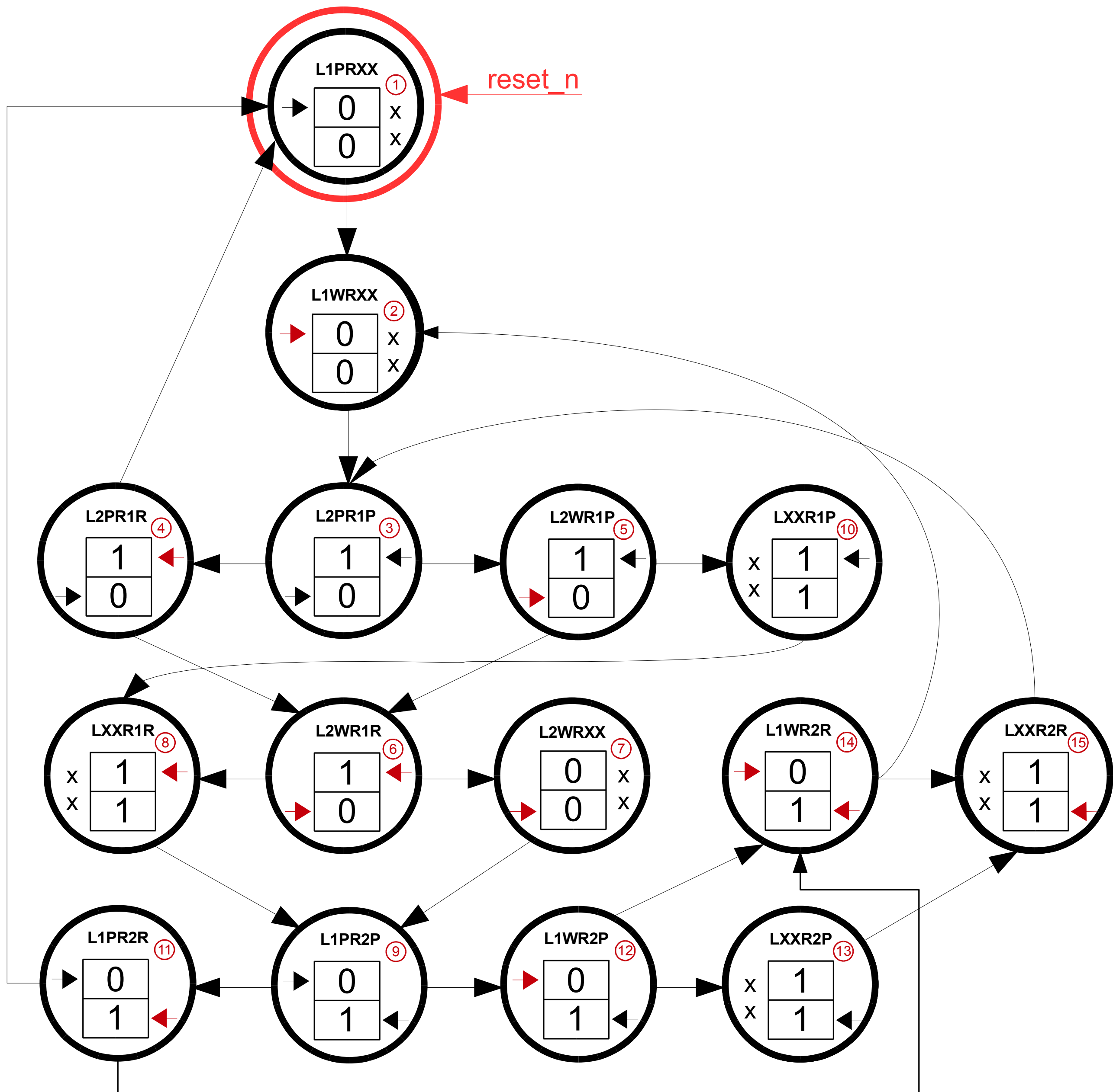
5.2 Vnitřní uspořádání

Blok GIB je navržen tak, aby mohlo každé z rozhraní pracovat na rozdílné frekvenci. GIB obsahuje dvouportovou paměť RAM, která umožňuje zároveň zapisovat i číst data. Zabraňování kolizí má na starosti řadič. Ten generuje korespondenční signály pro obě strany. Stará se také o řízení offsetu v adresách, čímž zabraňuje vzniku kolizí. Protože každá ze stran pracuje na jiné frekvenci, je nutno synchronizovat signály, které procházejí mezi jednotlivými hodinovými doménami. Je také nutno synchronizovat reset pro obě hodinové domény. Adresa je před pamětí RAM převáděna z binárního na Grayův kód, to má dopomoci ke snížení spotřeby [2]. Schéma vnitřního uspořádání bloku GIB je na obrázku 36. Signál *re*, který povoluje čtení, byl zaveden do multiplexoru. Původní řešení, kdy byl signál *re* zaveden do bloku RAM, vedlo ke vzniku kolizí v RAM (na adrese 0x00).

Ilustrace 36: Schéma vnitřního uspořádání bloku GIB

5.2.1 Řadič

Řadič se stará o generování korespondenčních signálů a o zabránění kolizí v přístupu k paměti RAM. Obsahuje celkem 15 stavů (viz obr. 37). Je implementován jako stavový automat typu Moore s registrovanými výstupy. Registrované výstupy mají eliminovat na výstupu z automatu zákmity.



Obrázek 37: Schéma stavového automatu v bloku GIB

Legenda k dékodování stavů automatu :
-- L,P - levá,pravá strana
-- 1,2 pozice v RAM
-- p - ukazatel
-- w,r operace zápis,čtení
-- xx - není možno na dané straně provádět žádné operace (paměť je plná)
-- červené číslo v kroužku je číslo stavu

5.3 Verifikace bloku GIB

Verifikaci bloku GIB lze rozdělit do několika kroků:

1. RTL simulace řadiče
2. RTL simulace datové cesty
3. Hradlová simulace bloku GIB
4. Hradlová simulace bloků GIB v sérii s *dummy* bloky a hraničními bloky
5. Hradlová simulace připojení výpočetní posloupnosti k procesoru OR1200

Při verifikaci byl použit blok pro generování hodin převzatý z [18].

5.3.1 RTL simulace řadiče a datové cesty

Tato simulace má za úkol prověřit správné fungování všech komponent v datové cestě (úpravy adresy, chování paměti RAM).

5.3.2 Hradlová simulace bloku GIB

Blok GIB obsahuje dvouportovou paměť RAM, kdy každá strana může pracovat s rozdílnou frekvencí hodin. Verifikace bloku GIB probíhá v režimu kontinuálního zapisování a čtení (obě operace jsou prováděny současně). Test bloku GIB vypadá tak, že je celá paměť bloku GIB zaplněna posloupností čísel a na výstupu musí být tato posloupnost čísel přečtena v nezměněném pořadí. Tento zápis a čtení je pak prováděn nejméně pro 30 zápisů a čtení bezprostředně za sebou. Verifikaci bloku GIB je nutno provést pro různé frekvence vstupních hodin. Je nutno pokrýt všechny možnosti:

1. $f_1 = f_2$
2. $f_1 < f_2$ kdy verifikace je provedena pro soudělné a nesoudělné hodnoty frekvencí
3. $f_1 > f_2$ kdy verifikace je provedena pro soudělné a nesoudělné hodnoty frekvencí

5.3.3 Hradlová simulace bloků GIB v sérii s *dummy* bloky a hraničními bloky

Protože bloky GIB budou připojeny k procesoru OpenRISC pomocí sběrnice Wishbone je nutno verifikovat chování bloků GIB spolu hraničními bloky, které budou realizovat most mezi GIB a Wishbone. Protože zatím není k dispozici žádné DSP jádro, je nutno připravit na RTL úrovni v jazyce VHDL *dummy* blok, který bude pouze překopírovávat data z jednotlivých bloků GIB.

Dummy blok a část bloků GIB může pracovat na jiné frekvenci než hraniční bloky. Verifikace bude probíhat obdobně jako test samotného bloku GIB s tím rozdílem, že data budou zapisována a čtena přes sběrnici Wishbone.

5.4 Implementace bloku GIB

Blok GIB je syntetizován na obvod Spartan 6 (xc6slx45-3csg324) pro pouzdro CSG324, speed grade -3. Samotný blok může podle STA (*Static Time Analysis*) pracovat na frekvenci 239 MHz. Důležitější však bude jaká bude maximální pracovní frekvence s hraničními bloky. Během syntézy bloku se nevyskytla žádná varování. Makro HDL statistiky pro šířku adresy 8 bitů jsou uvedeny v tabulce 15.

Použitá makra	Použitá primitiva na FPGA
# RAMs : 1 512x32-bit dual-port block RAM : 1 # Registers : 14 Flip-Flops : 14 # Multiplexers : 1 8-bit 2-to-1 multiplexer : 1 # FSMs : 1 # Xors : 2 9-bit xor2 : 2	# BELS : 29 # GND : 1 # INV : 3 # LUT2 : 8 # LUT3 : 8 # LUT4 : 4 # LUT6 : 4 # VCC : 1 # FlipFlops/Latches : 18 # FDC : 17 # FDP : 1 # RAMS : 1 # RAMB16BWER : 1 # Clock Buffers : 2 # BUFGP : 2

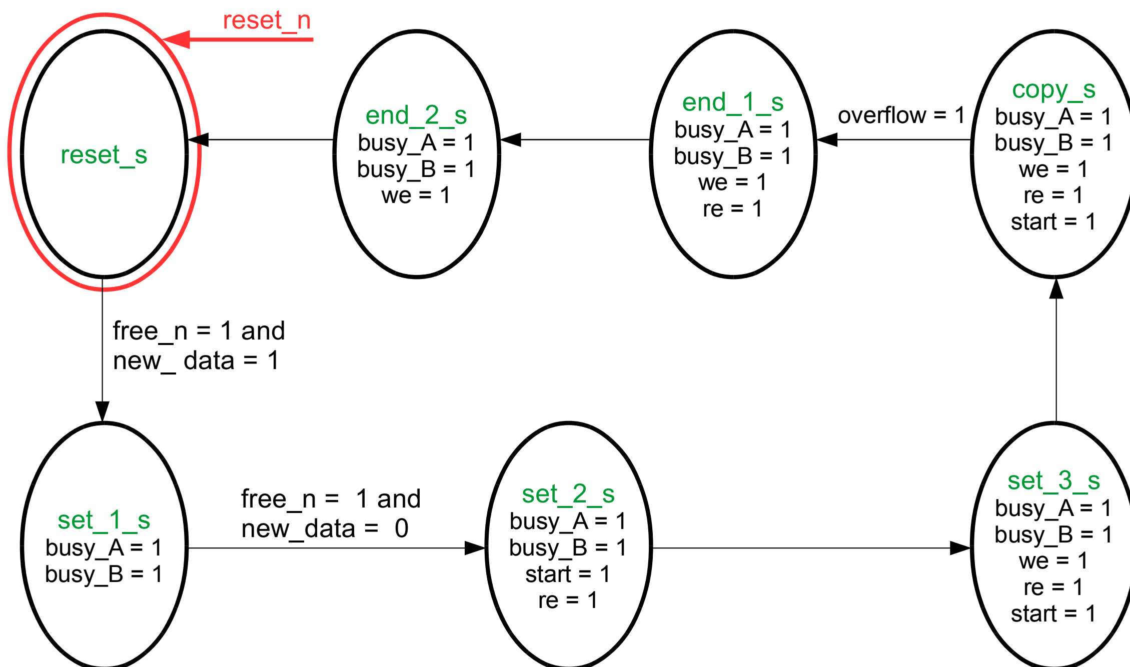
Tabulka 15: Použitá HDL makra v GIB

5.5 Návrh *dummy* bloku

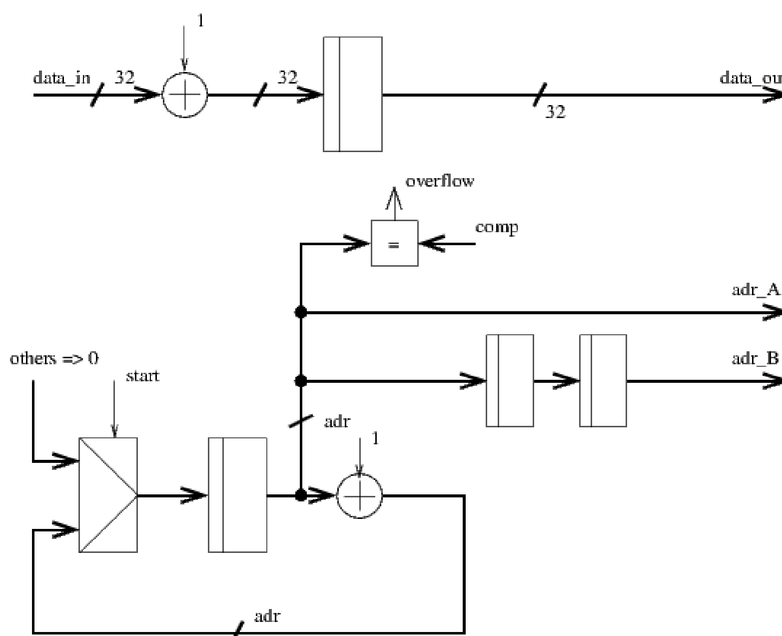
Dummy blok je navržen proto, aby umožnil otestovat a zapojit bloky GIB. Tento blok se zapojí místo DSP bloku (viz obr. 31) a pouze překopíruje data z jednoho bloku GIB do druhého. Tento blok je navržen na úrovni RTL a je plně syntetizovatelný na FPGA. *Dummy* blok na jedné straně vyčítá data z GIBu a na druhé straně je ukládá do dalšího GIBu. Verifikace bloku je provedena spolu s verifikací bloku GIB a hraničními bloky.

5.5.1 Vnitřní uspořádání bloku *dummy*

Blok *dummy* je rozdělen na datovou cestu a řadič. Řadič se stará o obsluhu korespondenčních signálů z bloků GIB a spouští proces kopírování. Blok obsahuje čítač, který generuje adresu odkud jsou data čtena a kam jsou zapisována. Pokud čítač dosáhne určité hodnoty, je kopírování ukončeno. K datům, která jsou přečtena, je jako kontrola, že byla překopírována, přičítána 1. Blok *dummy* je nastaven tak, aby překopíroval celý obsah paměti v bloku GIB. Schéma datové cesty *dummy* bloku je na obrázku 39, schéma řadiče je potom na obrázku 38. Protože adresa zapisovaných dat je o dva hodinové takty zpožděna za čtenými daty, jsou jako zpožďovací členy na signálu *adr_B* využity dva registry.



Obrázek 38: Schéma stavového automatu bloku dummy



Obrázek 39: Schéma datové cesty bloku dummy

5.6 Výsledky verifikací bloků GIB a dummy

Blok GIB byl otestován podle všech 5 bodů (viz. kap. 5.3) . S verifikací bloku GIB proběhla i verifikace bloku *dummy.* Během verifikace bylo nalezeno několik chyb (chyby ve stavovém automatu bloku GIB, chyby v datové cestě vedoucí ke kolizím v RAM). Všechny nalezené chyby byly opraveny. Oba bloky považuji verifikací za dostatečně pokryté.

6 Návrh mostu mezi Wishbone a GIB

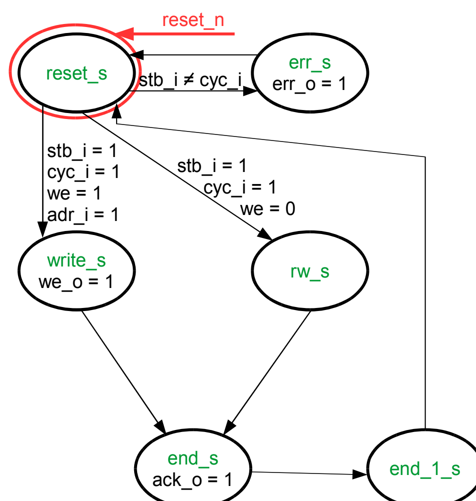
Hraniční bloky umožňují připojení celé výpočetní posloupnosti na sběrnici Wishbone (podle obrázku 24), skrze kterou k ní může přistupovat procesor OpenRISC. Jak je z obrázku patrné, existují dva typy mostů. Do jednoho jsou data nahrávána (*bridge master*), zatímco z druhého jsou data vyčítána (*bridge slave*). Kromě pouhého předávání dat ze sběrnice Wishbone do GIBu se hraniční blok musí také starat o management korespondenčních signálů bloku GIB (signály *busy*, *free_n*, *new_data*). Jako nejvýhodnější varianta se jeví ta, kdy hraniční blok obsahuje registry, které jsou přístupné na určité adrese, a skrze tyto registry je možné ovládat řídící signály bloku GIB. Zároveň je v takovém případě možné, aby hraniční blok generoval přerušení, pokud jsou k dispozici nová data (*bridge slave*) nebo je možné do výpočetní posloupnosti poslat nová data (*bridge master*). Detailní popis sběrnice Wishbone je v kapitole 4.3.1

6.1 Vnitřní uspořádání hraničních bloků

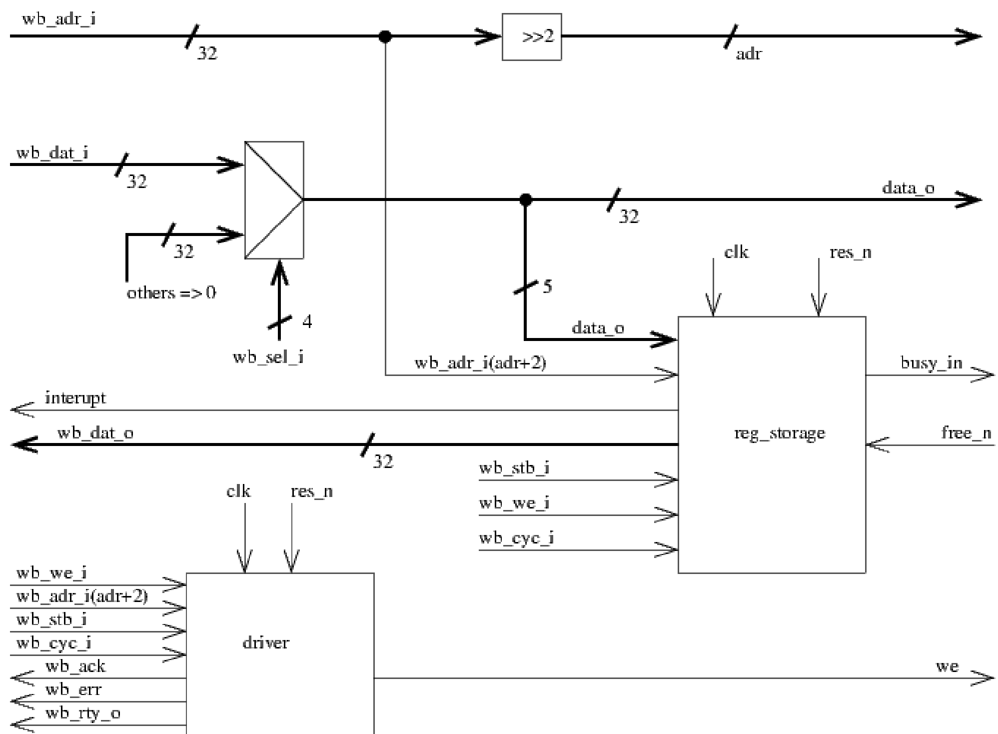
Protože každý most přistupuje k bloku GIB z různých stran (jeden čte a druhý zapisuje) (viz obr. 31), liší se i jejich stavové automaty a vnitřní struktura. Oba bloky obsahují shodný blok obsahující registry. Řadiče u obou bloků jsou implementovány jako automat typu Moore. Z důvodu eliminování zákmitů na výstupu ze stavového automatu je kódováni u obou automatů nastaveno na typ One-hot. U obou bloků musí být upravena vstupní adresa. Ta musí být vydělena 4 (dělení se provede bitovým posunem), protože OpenRISC pracuje pouze s adresou, která je násobkem 4.

6.1.1 Blok *bridge master*

Tento blok nahrává data do prvního GIBu ve výpočetní posloupnosti. Schéma jeho vnitřního uspořádání je na obrázku 41, řadiče potom na obrázku 40.



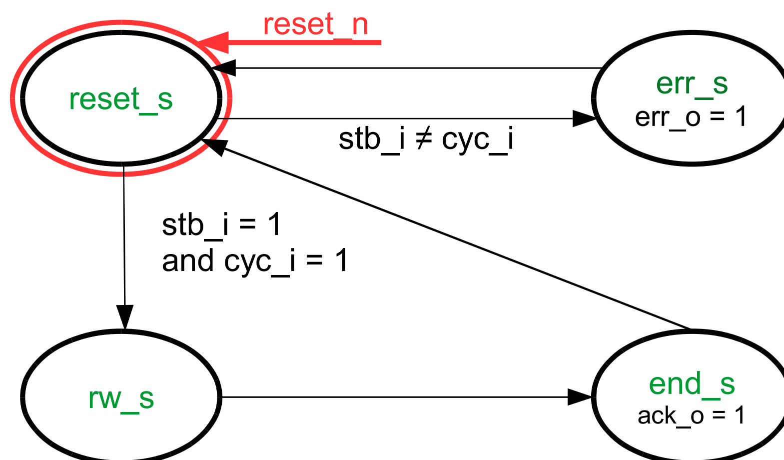
Obrázek 40: Schéma stavového automatu v bloku bridge master



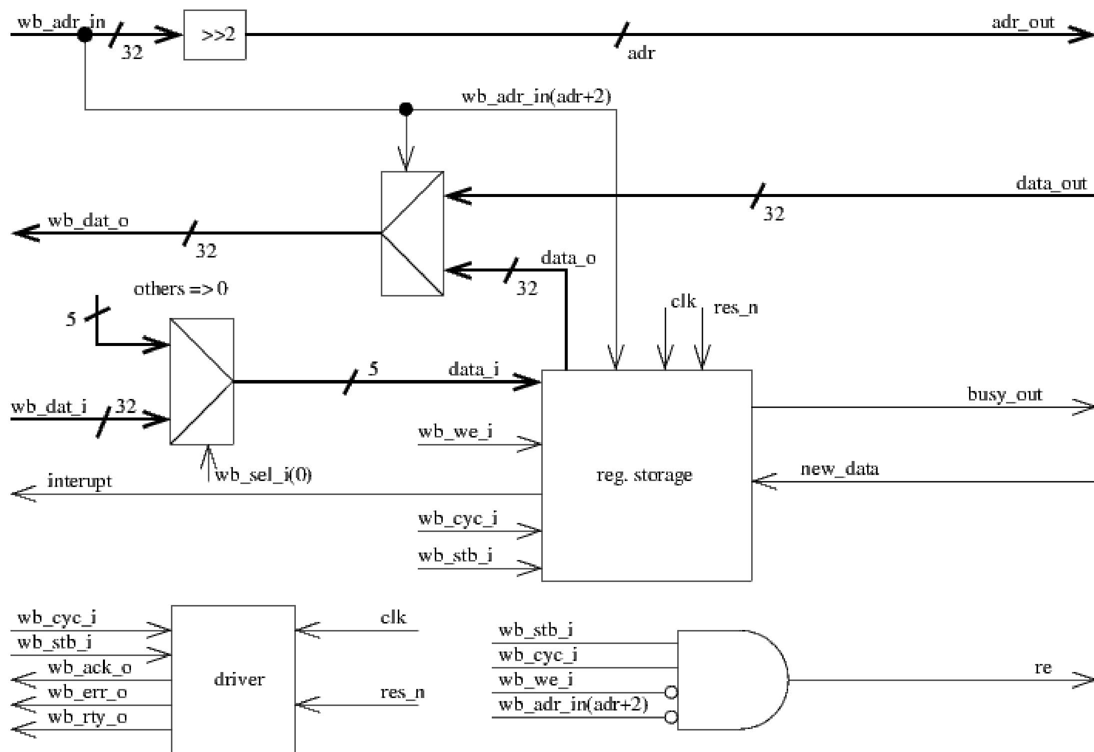
Obrázek 41: Vnitřní schéma bloku bridge master

6.1.2 Blok *bridge slave*

Tento blok vyčítá data z posledního GIBu ve výpočetní posloupnosti. Schéma jeho vnitřního uspořádání je na obrázku 43, řadiče potom na obrázku 42.



Obrázek 42: Schéma stavového automatu v bloku bridge slave



Obrázek 43: Vnitřní schéma bloku bridge slave

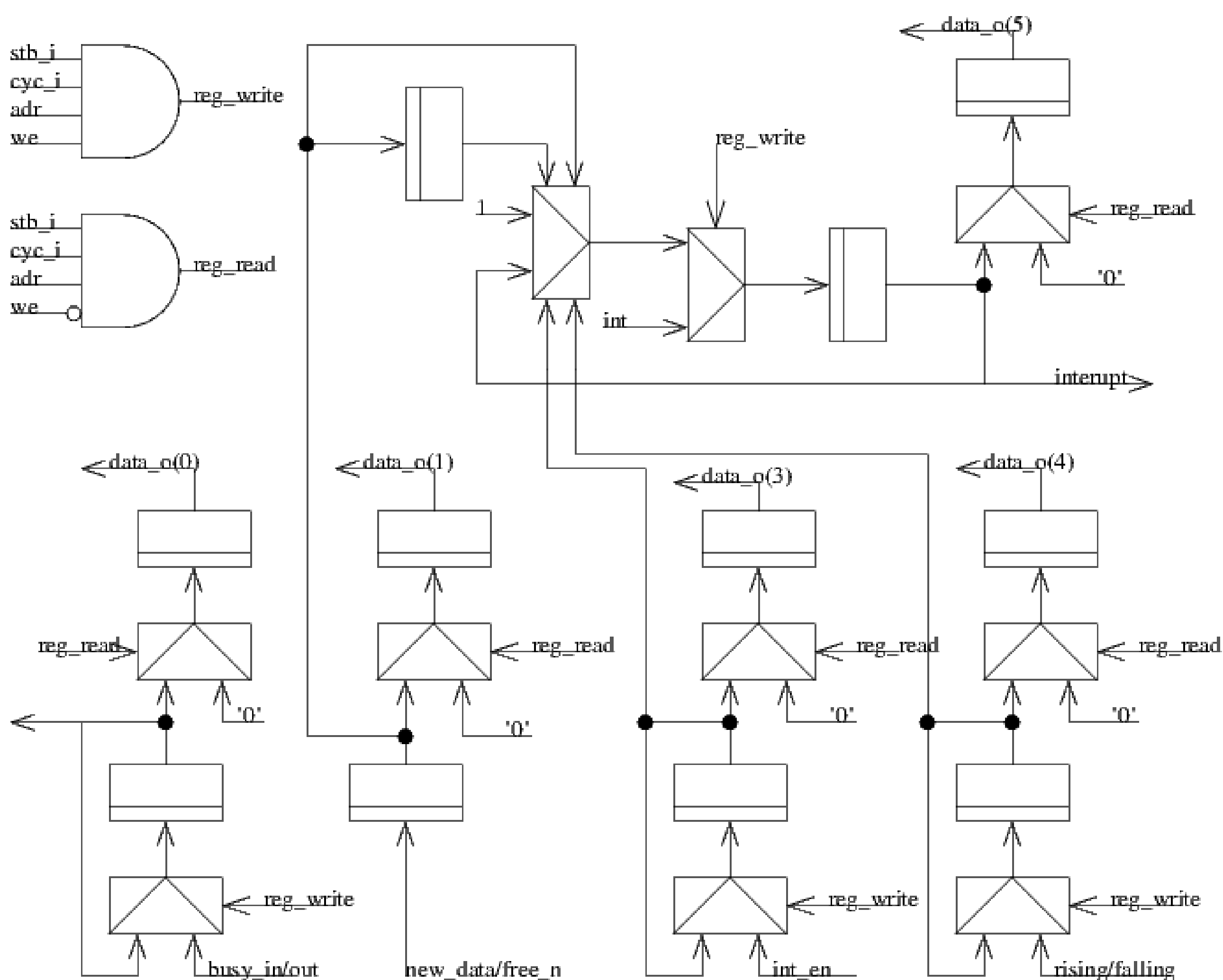
6.1.3 Pole registrů

Pole registrů (*register_field*) obsahují oba hraniční bloky. Tento blok obsahuje registry, skrze které je ovládán příslušný blok GIB a také generuje přerušení, pokud je do příslušného bloku GIB možné zapsat nová data nebo pokud jsou nová data k vyzvednutí na bloku GIB. Přístup k těmto registrům je na adrese o jedna větší než nejvyšší adresa, kam jsou zapisována samotná data. Pokud tedy jsou data do bloků GIB zapisována na relativní adrese 0x00 – 0xFF, jsou nastavovací registry přístupné na adrese 0x100. Popis jednotlivých bitů je v tabulce 16. Vnitřní schéma bloku pole registrů je pak na obrázku 44.

bit	0	1	2	3	4
	busy_in/ busy_out	free_n/ new_data	int_en	rising edge/ falling edge	int
poč. hodnota	0	-	0	0/1	0
Read/Write	R/W	R	R/W	R/W	R/W

Tabulka 16: Popis registrů v poli registrů

- **Bit 0 – busy_in/busy_out :** Slouží pro ovládání signálu *busy* pro příslušný připojený GIB v závislosti na tom, jestli se jedná o hraniční blok *master* (ovládá signál *busy_in*) nebo *slave* (ovládá sígnál *busy_out*).
- **Bit 1 – free_n/new_data :** Skrze tento bit je možno číst stav signálů free_n a new_data v závislosti na tom, jestli se jedná o hraniční blok *master* (signál *free_n*) nebo *slave* (signál *new_data*).
- **Bit 2 - int_en :** Bit slouží pro povolování přerušení. Pokud je nastaven do log. 1, je přerušení povoleno.
- **Bit 3 - rising edge/falling edge :** Tímto bitem se nastavuje, zda-li má být přerušení generováno při sestupné nebo vzestupné hraně signálů *free_n* nebo *new_data*. V závislosti na tom, jestli se jedná o blok *master* nebo *slave,* je (po resetu) nastavena výchozí hodnota log. 1 (*slave*) nebo log. 0 (*master*) . Tuto výchozí hodnotu je možno měnit zápisem. Hodnota log. 1 znamená, že se přerušení bude generovat na vzestupnou hranu a log. 0, že se přerušení bude generovat na sestupnou hranu *free_n* nebo *new_data*.
- **Bit 4 – int :** Log. 1 v tomto bitu informuje o tom, že došlo k přerušení. Zápisem log. 0 se toto přerušení vymaže.



Obrázek 44: Vnitřní uspořádání bloku register_field

6.2 Verifikace hraničních bloků

Verifikaci lze opět rozdělit do několika kroků:

1. RTL simulace řadiče, datové cesty a pole registrů.
2. Hradlová simulace obou mostů.
3. Hradlová simulace obou mostů s připojením ke zbytku výpočetní posloupnosti.
4. Hradlová simulace celé výpočetní posloupnosti s připojením k procesoru OpenRISC.

Při verifikaci byl použit blok pro generování hodin převzatý z [18].

6.2.1 RTL simulace řadiče, datové cesty a pole registrů

Tyto simulace mají za cíl prověřit správné chování řadiče, datové cesty a pole registrů, které je přítomno v obou blocích *bridge.*

6.2.2 Hradlová simulace obou mostů

Hradlová simulace samotných mostů by měla prověřit zejména čtení a zápis do interních registrových polí, měla by prověřit vyvolávání přerušení a také časové průběhy výstupů na straně připojení k blokům GIB.

6.2.3 Hradlová simulace obou mostů s připojením ke zbytku výpočetní posloupnosti

Tento způsob verifikace ověří funkčnost celé výpočetní posloupnosti. Tato verifikace je také probírána v kapitole 5.3.3. Verifikace by měla být provedena jednak s použitím přerušení a jednak bez použití přerušení kdy, stav bloků GIB bude čten přes registrové pole.

6.3 Implementace

Celý návrh (GIB, hraniční bloky, Dummy blok) je syntetizován na obvod Spartan 6 (xc6slx45-3csg324) pro pouzdro CSG324, speed grade -3. Podle STA může celá výpočetní posloupnost pracovat na maximální frekvenci 230,15 Mhz. Makro HDL statistiky celé výpočetní posloupnosti (pro velikost genericky nastavitelné adresy v bloku GIB na 8 bitů) a využití prostředků na FPGA je shrnuto v tabulce 17.

Použitá makra	Použitá primitiva na FPGA

<pre># RAMs : 2 	
 512x32-bit dual-port block RAM   : 2
# Adders/Subtractors : 1 	
 32-bit adder   : 1
# Counters : 1 	
 8-bit up counter   : 1
# Registers : 100 	
 Flip-Flops   : 100
# Multiplexers : 38 	
 1-bit 2-to-1 multiplexer   : 32
 32-bit 2-to-1 multiplexer   : 1
 5-bit 2-to-1 multiplexer   : 3
 8-bit 2-to-1 multiplexer   : 2
# FSMs : 5 	
# Xors : 4 	
 9-bit xor2   : 4</pre>
	<pre># BELS : 269
	# GND : 1
	# INV : 4
	# LUT1 : 30
	# LUT2 : 79
	# LUT3 : 21
	# LUT4 : 18
	# LUT5 : 19
	# LUT6 : 18
	# MUXCY : 38
	# VCC : 1
	# XORCY : 40
	# FlipFlops/Latches : 125
	# FDC : 115
	# FDCE : 5
	# FDP : 4
	# FDPE : 1
	# RAMS : 2
	# RAMB16BWER : 2
	# Clock Buffers : 2
	# BUFGP : 2</pre>

Celkové využití prostředků FGPA	
Počet Slice Registers	125 z 54 576
Počet Slice LUTs	141 z 27 288
Použito jako logika (LUTs)	134 z 27 288
Počet obsazených Slices	79 z 6 822
Počet pamětí RAMB16BWER	2 z 116

Tabulka 17: Využití Spartanu 6 (xc6slx45-3csg324) výpočetní posloupností.

6.3.1 Varování

Během syntézy se vyskytlo několik varování (pod každým varování je vysvětlení):

WARNING:Xst:647 - Input <wb_adr_i<1:0>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.
WARNING:Xst:647 - Input <wb_adr_i<31:11>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.

Syntézní nástroj varuje, že signály nejsou zapojené. Je to z toho důvodu, že GIB, který je na tyto signály připojen, má genericky nastavenou adresu na 8 bitů a ostatní bity v adrese jsou tudíž nepoužívané.

WARNING:Xst:647 - Input <wb_dat_i<31:5>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.
WARNING:Xst:647 - Input <wb_sel_i<3:1>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.

Syntézní nástroj opět varuje, že signály nejsou zapojené. Zápis dat do tohoto bloku se děje pouze do pole registrů (data jsou zde z GIBu pouze čtena). Protože je v poli registrů pouze 5 registrů, ostatní datové signály nejsou používány.

WARNING:Xst:647 - Input <data_i<1:1>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.

Na pozici 1 v registrovém poli je vstup z bloku GIB (*new_data*). Protože tento signál je pouze pro čtení, není zápis do tohoto registru zapojen.

WARNING:Xst:2404 - FFs/Latches <data_o_q<31:5>> (without init value) have a constant value of 0 in block <reg_storage_1>.

Stejná situace v opačném směru. Registrů ze kterých jsou data čtena je 5. Sběrnice Wishbone však pracuje s 32 bitovými signály. Na ostatní signály je trvale nastavena log. 0.

WARNING:Xst:647 - Input <wb_adr_i<1:0>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.
WARNING:Xst:647 - Input <wb_adr_i<31:11>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.
WARNING:Xst:647 - Input <data_i<1:1>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.
WARNING:Xst:2404 - FFs/Latches <data_o_q<31:5>> (without init value) have a constant value of 0 in block <reg_storage_2>.

Stejné situace jako v prvním hraničním bloku.

6.4 Výsledky verifikace

Verifikace hraničních bloků byla provedena v plném rozsahu všech 4 bodů (viz kapitola 6.2). Během verifikace bylo nalezeno několik chyb (nešlo zapisovat do registrového pole, chyby ve stavových automatech), všechny nalezené chyby byly odstraněny. Pokrytí hraničních bloků verifikací považuji za dostatečnou.

7 Připojení výpočetní posloupnosti k procesoru OpenRISC

Jak bylo řečeno v kapitole 4.3, procesor OpenRISC je vybaven samostatnou instrukční a datovou sběrnicí typu Wishbone. Aby bylo na tuto sběrnici možno zapojit více bloků a nedocházelo ke kolizím, je nutno použít arbitr sběrnice (viz kapitola 4.3.1). Tento arbitr provádí dekódování adresy, kterou generuje Master, a podle této adresy ho spojí s příslušným blokem Slave. Arbitr naštěstí není potřeba navrhovat, komunita kolem procesoru OpenRISC dává k dispozici generátor arbitru [14] . Kromě sběrnice Wishbone je nutné na procesoru také korektně zapojit hodinový signál, reset, dále signály určené pro obsluhu přerušení, debugování a řízení spotřeby.

7.1 Konfigurace arbitru

Generátor arbitru se skládá ze skriptu v jazyce Perl (*wishbone.pl*) a konfiguračního souboru (*wishbone.defines*). Celý konfigurátor je možno spustit příkazem :

perl wishbone.pl -nogui wishbone.defines

Konfigurační soubor se skládá z několika částí – nejprve obecná konfigurace (typ multiplexorů, typ arbitru, cílové FPGA), konfigurace všech bloků typu Master a nakonec konfigurace všech bloků Slave. U všech bloků Slave se dá nastavit adresní prostor, jeho velikost a priorita přístupu. U mapování jednotek Slave, které obsahují periférie je potřeba dát pozor, aby byly namapovány do adresního prostoru, který není obsažen v cache. Nastavení tohoto adresního prostoru je potřeba překontrolovat v souboru *defines.v* , který je součástí nastavení procesoru OpenRISC. Mapování všech komponent Slave shrnuje tabulka 18. Soubor *wishbone.defines* je součástí přílohy na CD.

Komponenta	Paměťový rozsah	Cache
RAM	0x00000000 - 0x10000000	Ano
Uart	0x80000000 - 0x81000000	Ne
Bridge master	0x90000000 - 0x91000000	Ne
Bridge slave	0xA0000000 - 0xA1000000	Ne

Tabulka 18: Mapování komponent připojených na Wishbone

7.2 Sestavení testovací konfigurace

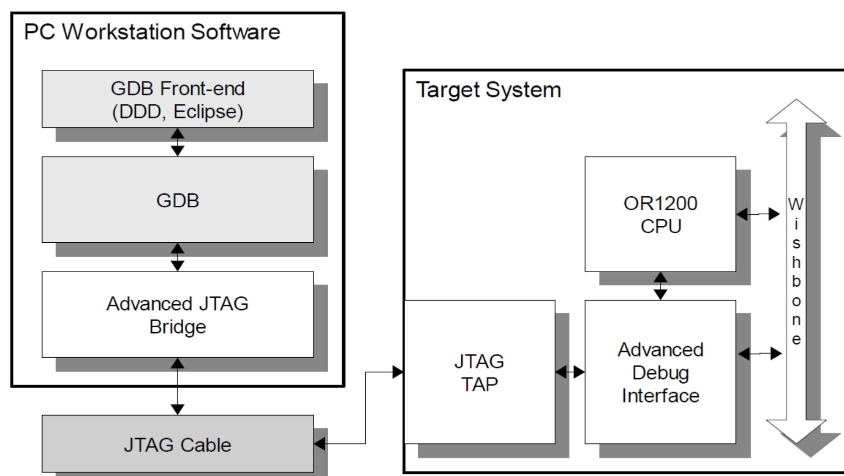
Pro verifikaci správné funkce hraničních bloků je nutné jejich připojení k procesoru OpenRISC. Správné zapojení procesoru OpenRISC lze popsat v několik krocích. Ideové schéma zapojení testovací konfiguace je na obr. 46.

- Zapojení hodin a resetu – Vzhledem k tomu, že deska ATLYS má pouze jeden 100 Mhz

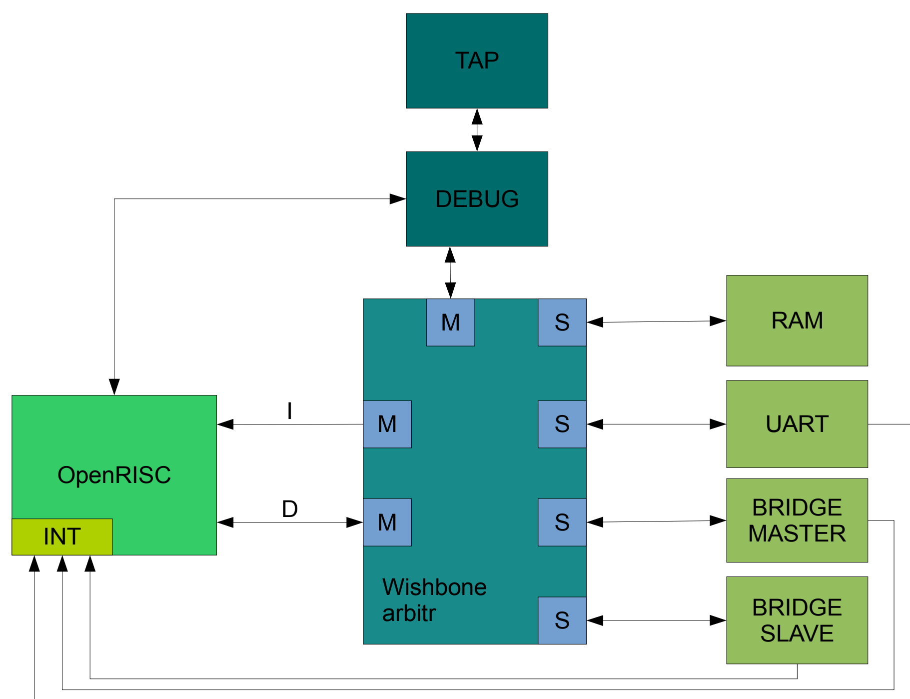
oscilátor a podle STA (*Static Time Analysis*) je maximální hodinová frekvence procesoru 61,23 Mhz, je nutno na FPGA využít specializovaný blok DCM (vytvoření hodinové děličky na FPGA se nedoporučuje), který sníží frekvenci v hodinovém stromě na FPGA na 50MHz. Reset je zapojen na resetovací tlačítko na kitu ATLYS. Reset je aktivní v log. 1.

- Připojení procesoru k arbitru sběrnice – Procesor OpenRISC je vybaven instrukční (jen pro čtení) a datovou (čtení a zápis) sběrnicí. Tyto sběrnice se zapojují právě na arbitr sběrnice.
- Zapojení přerušení – Všechny signály jednotek Slave, které generují přerušení (b*ridge master,bridge slave*, UART) je nutno zapojit na port *pic_ints.*
- Zapojení signálů pro debug - Protože procesor OpenRISC je vybaven podporou debugování, jako nejvýhodnější varianta se jevilo této podpory využít a za podpory použití bloku *Advanced debug interface* [16], který umožňuje nejen krokovat program, ale také ho nahrávat do paměti RAM. To vše za pomoci rozhraní JTAG, kterým je FPGA Spartan 6 vybaven. Skrze toto rozhraní je FPGA možné konfigurovat a také k němu mohou přistupovat komponenty umístěné uvnitř FPGA skrze rozhraní TAP. Je tedy možné skrze jeden JTAG konfigurovat jak samotné FPGA, tak nahrávat program a ladit jej uvnitř FPGA. I když tato možnost byla zprovozněna v práci [5], nepodařilo se mi ji znovu zprovoznit. Ideové schéma zapojení jednotky *Advanced debug interface* je na obrázku *45.* Pokud bychom jednotku *Advanced debug interface* nezapojovali*,* je nutno nastavit signál *dbg_stall_i* do log. 0.
- Zapojení řízení napájení - Řízení napájení je určené zejména pro případy, kdy je OpenRISC vyroben jako ASIC. Pro ostatní případy je důležité, aby signál *dbg_stall_i* byl v log. 0.

Program se nahrává do paměti RAM převzaté z [5], kterou je možno inicializovat buďto již během syntézy (což je však časově náročné) nebo například utilitou fy. Xilinx *data2mem* po PaR. Pro komunikaci s okolím je použita jednotka UART [15], která byla rovněž použita v [5].



Obrázek 45: Ideové schéma připojení debug jednotky. Převzato z [14].



Obrázek 46: Ideové schéma zapojení testovací konfigurace

7.3 Procesor z pohledu programátora

Jedná se o registrovou architekturu, kdy je k dispozici 32 třicetidvoubitových registrů, nad kterými jsou prováděny všechny operace. Některé registry mají speciální význam. Jejich shrnutí je v příloze. Procesor je ovládán sadou speciálních funkčních registrů (SFR), ke kterým je možno přistupovat specializovanými instrukcemi. Procesor má pevně dané vektory přerušení, které jsou rovněž shrnuty v příloze. Z našeho pohledu je nejdůležitější reset vektor a vektor externího přerušení, kde je zpracováváno přerušení z hraničních bloků. Několikrát jsem se také setkal s chybou zarovnání (všechny adresy na sběrnici musí být dělitelné 4).

7.4 GNU nástroje

Za hlavní výhodu procesoru OpenRISC lze považovat funkční port GNU nástrojů na tuto platformu.

Na webu OpenCores [17] je k dispozici ke stažení překladač, linker, C knihovna a také simulátor. Všechny nástroje je nutno zkompilovat. Postup kompilací a jednotlivé závislosti (před kompilerem je nutno např. zkompilovat simulátor) jsou na webu OpenCores [17]. Překladač neinicializuje automaticky ukazatel zásobníku, tudíž je třeba na reset vektor vložit krátký úsek kódu, který ho nastaví a poté provede skok na začátek programu. Rovněž všechny přerušení je třeba obsloužit v jazyce symbolických adres. Nejprve je třeba uložit obsah všech pracovních registrů do zásobníku, poté je možno provést odskok na funkci v jazyce C, která přerušení zpracuje. Následně je nutno

nahrát obsah pracovních registrů zpět. Rovněž přístup k SFR je možný pouze prostřednictvím jazyka symbolických adres (popřípadě použití příkazu A*SM* v C). Je také nutno ručně spravovat makefile.

7.5 Testovací program

Správnost připojení hraničních bloků k procesoru OpenRISC je nutno otestovat SW. Toto připojení jsem otestoval nejprve bez přerušení a poté s přerušením.

Test bez přerušení spočíval v nekonečné smyčce, která pravidelně zapisovala na adresy 0 až 256 v bloku GIB (maximální rozsah bloku GIB) čísla od 0 v sestupné posloupnosti a následně byla vyčtena z druhého hraničního bloku (brige slave). Poté byl proveden test, zda-li jsou čísla o jednu větší, než byla zapsána (ve výpočetní posloupnosti byl zapojen dummy blok) v případě, že bylo vše v pořádku je na UART vypsán řetězec „OK!“. V opačném případě je vypsán řetězec „FAIL!“.

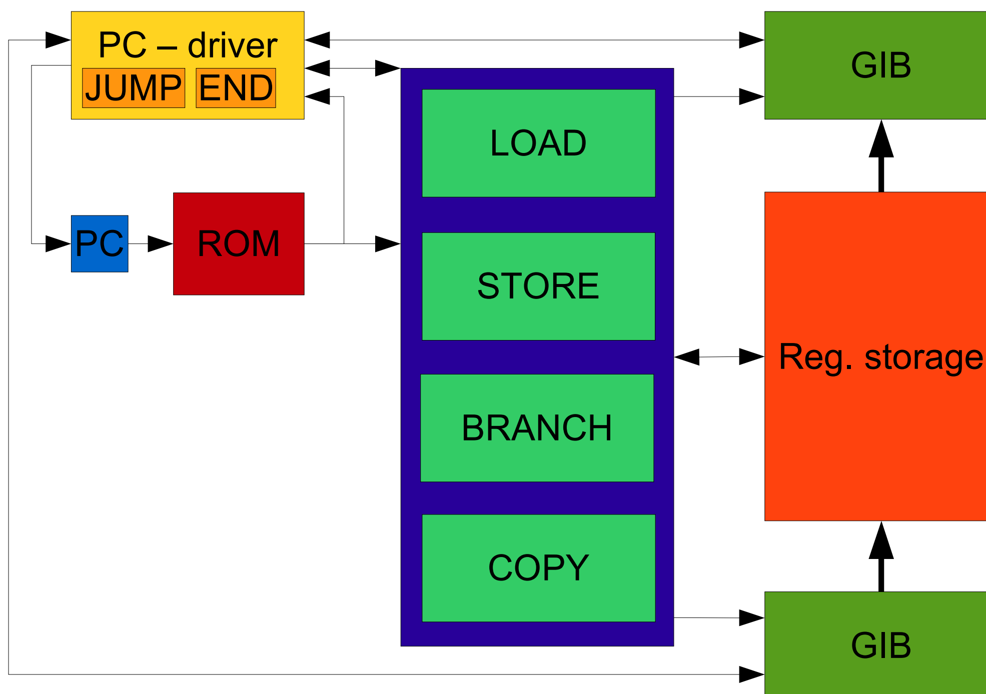
Test s přerušením spočívá stejně jako minulý test v zápisu čísel v sestupné posloupnosti s tím rozdílem, že kontrola dat je provedena pod přerušením vyvolaným hraničním blokem.

8 Výpočetní modul

V článku [3] byla nastíněna myšlenka, že by DSP bloky byly navrženy HLL syntézním nástrojem (byl uvažován Catapult C) . Za hlavní výhodu tohoto způsobu návrhu je považována jeho rychlost. Nicméně v průběhu vytváření této práce došlo ke změně licenční politiky a tento nástroj není v současnosti možné používat. Z tohoto důvodu bylo rozhodnuto vytvořit jednoduchou demonstrační výpočetní aplikaci na úrovni RTL. Jako demonstrační aplikaci by měl výpočetní blok realizovat jednoduchou funkci x^2 a jednoduchý filtr typu klouzavý průměr po dobu jedné sekundy pro vzorky ze všech kanálů. Pro demonstrační aplikaci jsou uvažovány 4 kanály. Původně byla tato demonstrační aplikace zamýšlena jako jeden velký stavový automat, který by obsloužil řídící signály z bloku GIB a nakopíroval data do příslušných výpočetních modulů. Tento přístup by však při jakékoliv změně (například přidání většího počtu kanálů) znamenal přepracování stavového automatu a jeho novou verifikaci. Proto jsem se rozhodl tento blok pojmout jako jednoduchý programovatelný automat se sadou základních příkazů pro práci s blokem GIB, kdy při změně například počtu kanálů bude stačit pouze upravit krátký program, ostatní parametry se nastaví generickými konstantami.

8.1 Návrh programovatelného DSP bloku

Programovatelný DSP blok musí mít schopnost číst jednotlivé položky z bloku GIB, nahrát je do samotných výpočetních bloků (např. filtr) a hotové výsledky uložit do paměti dalšího bloku GIB. Tyto výpočetní bloky jsou uloženy v registrovém poli, kde je možné mít jak obecné registry pro uložení mezivýsledků, tak je možné místo registrů na určité adresy namapovat výpočetní bloky. Musí umět rozpoznat, o jaký typ paketu se jedná (Load, Save, Reset). Navíc počet vzorků v paketu se může měnit, automat tedy musí mít i základní podporu větvení. Z těchto podmínek vyplývají potřebné příkazy (LOAD pro načtení položky z bloku GIB a její uložení do registrového pole, STORE pro uložení z reg. pole do GIB a příkaz BRANCH pro podporu větvení) . Programovatelný automat se musí spustit, pokud jsou k dispozici nová data a v následujícím bloku GIB je místo na další paket. Ideová struktura je na obrázku 47. PC (*Program counter*) má svůj vlastní řadič, který pokud jsou signály z bloků GIB *new_data* v log. 1 a *free_n* v log. 0, dojde ke spuštění čítání od adresy 0x00. Jednotlivé příkazy jsou uloženy v paměti ROM. Každá instrukce je představována jedním řadičem, který ji vykonává. Ten je spouštěn na základě kódu uloženého v paměti ROM. Po dobu vykonávání těchto instrukcí je PC zastaven.



Obrázek 47: Ideové schéma programovatelného DSP bloku

8.1.1 Registrové pole

Registrové pole je navrženo tak, aby kromě registrů pro obecné použití bylo možné na jednotlivé adresy namapovat výpočetní bloky, a to nejen vstup a výstup pro data, ale také nastavení jednotlivých bloků. Pole má samostatnou bránu pro čtení a zápis.

8.1.2 PC

PC je ovládán samostatným řadičem. Řadič pro PC rovněž ovládá signály *busy_in* a *busy_out.* Kvůli tomu také tento řadič implementuje příkaz pro ukončení práce End. Velikost PC je možno nastavit generickou konstantou.

8.1.3 ROM

V paměti ROM jsou uloženy všechny příkazy. Automat je nutno naprogramovat před syntézou. Velikost paměti je možno nastavit generickou konstantou. ROM je záměrně navržena tak, aby byla vysyntetizována z LUTů kvůli úspoře BRAM pro bloky GIB.

8.1.4 Jednotlivé příkazy

Kromě základních instrukcí LOAD, STORE a BRANCH bylo nutné instrukční sadu doplnit o další instrukce COPY (zkopíruje data mezi dvěma registry) a JUMP (provede skok na libovolnou adresu), END (ukončí výpočet, resetuje PC) a NOP (prázdná instrukce). Každá instrukce má 3

bitový opcode umístěný tak aby začínal v MSB. Délka všech instrukcí je stejná. Protože ne všechny instrukce mají stejný počet parametrů, je jejich délka dorovnána nulami. Délka těchto ostatních parametrů je nastavitelná generickými konstantami (adresy skoků – velikost PC, adresy registrů), proto se může měnit i celková délka instrukcí. Řadiče všech instrukcí a PC jsou Moorovy automaty s registrovanými výstupy [13].

<table>
<tr><th>Příkaz</th><th>Opcode</th><th colspan="5">Zbytek instrukce</th></tr>
<tr><th></th><th>MSB</th><th colspan="5">LSB</th></tr>
<tr><td>LOAD</td><td>0 1 0</td><td>Startovací bit</td><td colspan="3">nuly</td><td>Adresa cílového registru</td></tr>
<tr><td>STORE</td><td>0 1 1</td><td>Čekací bit</td><td colspan="3">nuly</td><td>Adresa ukládaného registru</td></tr>
<tr><td>BRANCH</td><td>1 0 0</td><td>Minus bit</td><td colspan="2">Adresa registru k porovnání</td><td>Porovnávaná hodnota</td><td>Adresa skoku</td></tr>
<tr><td>COPY</td><td>1 0 1</td><td>Čekací bit</td><td>Startovací bit</td><td>nuly</td><td>Adresa cílového registru</td><td>Adresa ukládaného registru</td></tr>
<tr><td>JUMP</td><td>1 1 0</td><td colspan="4">nuly</td><td>Adresa skoku</td></tr>
<tr><td>END</td><td>0 0 1</td><td colspan="5">nuly</td></tr>
<tr><td>NOP</td><td>0 0 0</td><td colspan="5">nuly</td></tr>
</table>

Tabulka 19: Popis kódu jednotlivých instrukcí

LOAD

Nahraje data z GIB a uloží je do registrového pole. Instrukce má vlastní čítač adresy pro čtení z GIB, který se po každém čtení automaticky inkrementuje. Tento čítač se při ukončení relace (příkaz END) resetuje. V příkazu se jedním bitem zapíná výpočetní operace (start), pokud je zápis prováděn do výpočetního bloku (filtr) Pokud je tento bit v log.1 tak instrukce čeká dokud se předchozí operace neukončí a poté provede zápis. Opcode a popis instrukce je v tabulce 19.

STORE

Uloží data z registrového pole do bloku GIB. Instrukce má stejně jako LOAD svůj vlastní čítač. Ten se opět při ukončení operací (příkaz END) vynuluje. V příkazu se jedním bitem zapíná čekání na dokončení výpočetní instrukce. Pokud je tento bit zapnut, instrukce čeká, dokud se výpočetní operace (např. filtr) namapovaná v registrovém poli neukončí. Popis instrukce je v tabulce 19.

BRANCH

Instrukce pro podporu větvení. Instrukce porovná hodnotu v kódu instrukce s hodnotou v

registrovém poli a pokud se shodují, provede skok. V instrukci je bit, který pokud je v log. 1, tak pokud se hodnoty neshodují, odečte 1 z hodnoty uložené v registru.

COPY

Instrukce překopíruje data mezi dvěma registry v registrovém poli. V instrukci je jedním bitem opět možno nastartovat výpočetní operaci (v případě zápisu do výpočetního bloku). Pokud je tento bit nastaven tak je opět čekáno neskončí předchozí výpočet. Dalším bitem zapnout čekání na dokončení operace při zápisu.

JUMP

Instrukce provede skok na zadanou adresu. Instrukce je vykonávána řadičem PC.

END

Instrukce ukončí operace nad oběma bloky GIB, resetuje čítače instrukcí LOAD a SAVE. Tato instrukce je vykonávána řadičem PC.

NOP

Prázdná instrukce. Protože je při resetu PC nastaven na nulu, musí být na adrese 0x00 uložena instrukce NOP, aby automat neprováděl žádnou činnost.

8.2 verifikace

Verifikaci lze opět rozdělit do několika bodů:

1. RTL verifikace jednotlivých příkazů
2. RTL verifikace posloupnosti příkazů
3. Hradlová simulace posloupnosti příkazů
4. Hradlová simulace posloupnosti příkazů s připojením k blokům GIB a hraničním blokům

Cílem verifikací z bodu 1 je na RTL úrovni ověřit správné fungování jednotlivých instrukcí pro všechny varianty těchto instrukcí (zapnutý minus bit apod.). V bodě 2 a 3 je potom cílem těchto simulací ověřit správnou funkci příkazů, které jsou řazeny postupně za sebou. Příkazy jsou vybrány tak, aby jejich správnost bylo možno ověřit podle výsledků, které bude DSP blok ukládat do bloku GIB. V bodě 4 pak dojde k otestování připojení tohoto bloku k blokům GIB spolu s hraničními bloky. Při verifikaci byl použit blok pro generování hodin převzatý z [18].

Jako posloupnost příkazů pro body 2,3 a 4 jsem zvolil příkazy:

```
END_OP&END_P,                 --7
JUMP_OP&JM&x"02",             --6
BRANCH_OP&'1'&R0&x"00"&x"07", --5
STORE_OP&'0'&LS&R3,           --4
COPY_OP&'0'&'0'&CP&R3&R1,     --3
LOAD_OP&'0'&LS&R1,            --2
LOAD_OP&'0'&LS&R0,            --1
NOP_OP&NOP                    --0
```

Na první pozici v paměti je příkaz NOP, následován příkazem LOAD, který do registru R0 nahraje hodnotu, která bude porovnávána ve smyčce. Následují příkazy, které se budou opakovat ve smyčce – LOAD, COPY a STORE. Příkaz BRANCH vyhodnotí, zda-li se hodnota v registru R0 rovná 0. Pokud ano, provede skok na adresu 0x07. Tam se nachází příkaz END, který ukončí celou relaci. Pokud se hodnota v registru R0 nerovná 0, registr R0 je dekrementován a program normálně pokračuje na adresu 0x06, kde se nachází příkaz JUMP, který provede skok na adresu 0x02 a smyčka se opakuje. Testbench pro tento typ programu nahraje nejprve do DSP bloku velikost překopírovávané posloupnosti a následně ji posílá do DSP bloku. Na výstupu pak zkontroluje, zda-li se překopírovala správně. Tímto testem je na hradlové úrovni prokázáno fungující spojení s bloky GIB. V tomto testu však na hradlové úrovni nebylo ověřeno správné chování instrukce BRANCH bez aktivovaného minus bitu. Proto jsem navrhl ještě jeden test, který speciálně tuto funkčnost ověří. Tento test probíhá na hradlové úrovni bez připojení k blokům GIB. Kód v ROM pro tento test je :

```
END_OP&END_P,                 --7
STORE_OP&'0'&LS&R0,           --6
STORE_OP&'0'&LS&R1,           --5
STORE_OP&'0'&LS&R0,           --4
BRANCH_OP&'0'&R0&x"08"&x"05", --3
LOAD_OP&'0'&LS&R1,            --2
LOAD_OP&'0'&LS&R0,            --1
NOP_OP&NOP                    --0
```

V tomto testu je nejprve nahrána do registru R0 hodnota 0x08. Následně do registru R1 libovolná hodnota. Pokud test proběhne v pořádku, instrukce BRANCH provede skok na adresu 0x05 a hodnoty ukládané do DSP bloku vypíše beze změny v opačném pořadí.

8.3 Návrh filtru

Jako demonstrační výpočetní aplikace byla zvolen výpočet $y_i = \frac{\sum_{j=0}^{n-1} x_j^2}{n}$ kde y_i je i-tý vzorek spočtený pro předchozích n-1 vstupních vzorků. Filtr klouzavý průměr je implementován jako dlouhý posuvný registr pro všechny vzorky, kde výstupy ze všech registrů jsou zavedeny do sčítaček. Pokud je počet registrů (vzorků) roven mocnině dvou je možno dělení provést jednoduše

jako bitový posun, což je v tomto návrhu využito. Filtr je navržen jako generický, kdy je možno měnit počet vzorků, nad kterými průměruje. Filtr je rozdělen na datovou a řídící část. Protože výstupem je při 16 bitovém vstupu 32 bitový výstup, rozhodl jsem se, že výstup bude oříznut na 16 bitů.

8.3.1 Verifikace filtru

Při verifikaci je nutno ověřit jednak správnou funkci filtru a také možnost napojení na registrové pole v DSP bloku. Správné fungování filtrů bylo ověřeno při nastavení délky průměrované posloupnosti na 4 vzorky a pro 16 bitové slovo. Funkce filtru byla ověřena pro následující vstupní data:

- Vstupní data rovna nule
- Vstupní data rovna maximálnímu rozsahu vstupního slova (65 535)
- Náhodná vstupní data malého rozsahu (hodnoty okolo 255)
- Náhodná vstupní data velkého rozsahu (hodnoty okolo 30 000)

Připojení k programovatelnému DSP bloku bylo ověřeno na následující sekvenci vstupního kódu:

```
END_OP&END_P,                  --8
STORE_OP&'1'&LS&FB2,           --7
COPY_OP&'1'&'1'&CP&FB2&FB1,    --6
STORE_OP&'1'&LS&FB1,           --5
LOAD_OP&'1'&LS&FB1,            --4
LOAD_OP&'1'&LS&FB1,            --3
LOAD_OP&'1'&LS&FB1,            --2
LOAD_OP&'1'&LS&FB1,            --1
NOP_OP&NOP                     --0
```

Výstupem tohoto testu jsou dvě hodnoty z filtrů v registrovém poli. Protože výše zmíněná sekvence kódu neotestovala všechny možnosti instrukce COPY, je nutno provést ještě jeden test, který tyto možnosti ověří. Jeho kód je :

```
END_OP&END_P,                  --5
STORE_OP&'0'&LS&R1,            --4
COPY_OP&'1'&'0'&CP&R1&FB1,     --3
COPY_OP&'0'&'1'&CP&FB1&R0,     --2
LOAD_OP&'0'&LS&R0,             --1
NOP_OP&NOP);                   --0
```

Zde jsou opět výstupem data z filtru.

8.3.2 Implementace

Filtr je syntetizován na obvod Spartan 6 (xc6slx45-3csg324) pro pouzdro CSG324, speed grade -3.

Výsledky STA jsou závislé na délce filtru. Výsledky HDL syntézy jsou shrnuty v tabulce 20 (pro 16 bitové vstupní slovo a průměrování pro 4 vzorky).

Použitá makra		Použitá primitiva na FPGA	
# Multipliers	: 1	# BELS	: 220
16x16-bit multiplier	: 1	# GND	: 1
# Adders/Subtractors	: 3	# INV	: 1
33-bit adder	: 1	# LUT1	: 1
35-bit adder	: 2	# LUT2	: 33
# Registers	: 163	# LUT3	: 35
Flip-Flops	: 163	# LUT4	: 32
# FSMs	: 1	# MUXCY	: 66
		# VCC	: 1
		# XORCY	: 50
		# FlipFlops/Latches	: 165
		# FDC	: 4
		# FDCE	: 160
		# FDP	: 1
		# Clock Buffers	: 1
		# BUFGP	: 1
		# DSPs	: 1
		# DSP48A1	: 1

Tabulka 20: detaily implementace demonstrační DSP funkce

Varování

Během syntézy se vyskytlo několik varování:

```
WARNING:Xst:2677 - Node <reg_out_q_0> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_1> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_2> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_3> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_4> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_5> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_6> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_7> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_8> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_9> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_10> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_11> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_12> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_13> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_14> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_15> of sequential type is unconnected in block <i_filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_0> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_1> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_2> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_3> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_4> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_5> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_6> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_7> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_8> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_9> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_10> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_11> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_12> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_13> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_14> of sequential type is unconnected in block <filter_data_path>.
WARNING:Xst:2677 - Node <reg_out_q_15> of sequential type is unconnected in block <filter_data_path>.
```

Tato varování jsou způsobena tím, že je výstupní slovo oříznuto na horních 16 bitů (syntetizátor hlásí, že dolních16 bitů z výstupu není použito).

8.3.3 Výsledky verifikace

Verifikace filtru byla provedena podle kapitoly 8.3.1 . Problémy se správnou funkcí filtru (filtr

dával špatné výsledky) byly odstraněny. Nicméně při STA bylo zjištěno, že se vzrůstající délkou filtru velmi narůstá nejdelší kombinační cesta v obvodu. Proto jsem do návrhu filtru vložil čekací stavy. Pokud by však byl filtr navržen pro zpracovávání delší časové posloupnosti a pro vyšší vzorkovací frekvenci (např. 1024 Hz), bylo by nutné tyto čekací stavy rozšířit, popřípadě do obvodu vložit čítač, který by odměřoval čas potřebný pro dokončení výpočtu. Protože jednotlivé vzorky jsou uloženy v samostatných registrech, pro větší zpracovávané posloupnosti velmi roste spotřeba registrů na FPGA.

8.4 Implementace

Programovatelný DSP blok je syntetizován na obvod Spartan 6 (xc6slx45-3csg324) pro pouzdro CSG324, speed grade -3. Protože je v obvodu umístěna paměť ROM, kterou není možno programovat za běhu, syntetizátor (jako vývojové prostředí na všechny bloky bylo používáno volně dostupné ISE Webpack fy. Xilinx) provádí analýzu kódu, který je v ní uložen, a objekty, které nejsou programem použity, vyřazuje z návrhu. Proto mohou v závislosti na použitých instrukcích kolísat výsledky STA (bude se měnit nejdelší kombinační cesta). Podle STA pro program z prvního testu (program bez zápisu do filtrů) může programovatelný DSP blok běžet na maximální frekvenci 157.23MHz. Výsledky HDL syntézy jsou shrnuty v tabulce 21.

<table>
<tr><th>Použitá makra</th><th>Použitá primitiva na FPGA</th></tr>
<tr><td><pre>
RAMs   : 1
 16x28-bit single-port distributed Read Only RAM : 1
Multipliers   : 4
 16x16-bit multiplier   : 4
Adders/Subtractors   : 14
 32-bit subtractor   : 1
 33-bit adder   : 4
 35-bit adder   : 8
 8-bit adder   : 1
Counters   : 2
 8-bit up counter   : 2
Registers   : 874
 Flip-Flops   : 874
Comparators   : 1
 8-bit comparator equal   : 1
Multiplexers   : 35
 1-bit 2-to-1 multiplexer   : 11
 1-bit 6-to-1 multiplexer   : 1
 1-bit 8-to-1 multiplexer   : 1
 16-bit 2-to-1 multiplexer   : 4
 32-bit 2-to-1 multiplexer   : 6
 8-bit 2-to-1 multiplexer   : 12
FSMs   : 9
</pre></td><td><pre>
BELS   : 694
GND   : 1
INV   : 3
LUT2   : 35
LUT3   : 220
LUT4   : 137
LUT5   : 28
LUT6   : 173
MUXCY   : 45
MUXF7   : 3
VCC   : 1
XORCY   : 48
FlipFlops/Latches   : 165
FDC   : 33
FDCE   : 132
Clock Buffers   : 1
BUFGP   : 1
</pre></td></tr>
</table>

Tabulka 21: Výsledky HDL syntézy programovatelného DSP bloku.

8.4.1 Varování

Během syntézy se vyskytlo několik varování :

```
WARNING:Xst:647 - Input <adress<7:4>> is never used. This port will be preserved and left
unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of
this sub-block is preserved.
```

Toto varování nás informuje o tom, že adresa má příliš mnoho bitů a vyšší bity nejsou využity.

```
WARNING:Xst:2677 - Node <reg_out_q_0> of sequential type is unconnected in block
<i_filter_data_path>.
```

Ořez datového slova u filtrů (toto varování se vyskytuje vícekrát pro všechny filtry).

```
WARNING:Xst:2677 - Node <pc_q_4> of sequential type is unconnected in block <top_compute_modul>.
WARNING:Xst:2677 - Node <pc_q_5> of sequential type is unconnected in block <top_compute_modul>.
WARNING:Xst:2677 - Node <pc_q_6> of sequential type is unconnected in block <top_compute_modul>.
WARNING:Xst:2677 - Node <pc_q_7> of sequential type is unconnected in block <top_compute_modul>.
```

Velikost PC byla nastavena na 8 bitů, rozsah adresy paměti ROM jsou však jen 4 bity – vyšší bity PC jsou proto nevyužity.

```
WARNING:Xst:2040 - Unit regs: 33 multi-source signals are replaced by logic (pull-up yes): ….
```

Aby bylo možno navrhnout generické multiplexory (a snadno tak přidávat kanály do tohoto pole), bylo nutno použít třístavovou logiku. Vnitřní třístavovou logiku však Spartan 6 nepodporuje, proto musela být tato logika nahrazena. Toto nahrazení nemá vliv na funkčnost.

```
WARNING:Xst:1895 - Due to other FF/Latch trimming, FF/Latch
<i_regs/gen_filter[3].i_filters/i_filter_data_path/reg_in_q_13> (without init value) has a constant
value of 0 in block <top_compute_modul>. This FF/Latch will be trimmed during the optimization
process.
```

Jak bylo již řečeno výše, pokud je v návrhu přítomna paměť ROM, syntetizátor analyzuje její obsah a odstraňuje bloky, které nebudou použity. Syntetizátor tedy správně vyhodnotil, že do filtrů nebude zapisováno ani z nich čteno, a proto je při syntéze odstranil. Těchto varování je v syntézním logu více.

8.5 Výsledky verifikace

Verifikace byla provedena podle kapitoly 8.2 . Funkčnost všech instrukcí programovatelného bloku DSP byla ověřena. Byl ověřen i zápis a čtení z výpočetních bloků umístěných v registrovém poli. Pokrytí všech instrukcí verifikací považuji za dostatečné.

9 Závěr

Hlavním cílem této práce bylo provést systémový návrh výpočetní dlaždice podle článku [3], zachovat modularitu a kompatibilitu se stávajícím systémem BCI a implementovat generický výpočetní blok (GIB) a bloky nutné k jeho připojení k stávajícímu HW. Koncept generického výpočetního bloku navrženého v článku [3] byl vylepšen a navržen tak, aby snižoval co nejvíce spínací aktivitu a tím i spotřebu. Kromě samotného generického výpočetního bloku byly navrženy i hraniční bloky umožnující připojení skrze sběrnici Wishbone k stávajícímu HW, který byl rozpracován v práci [5]. Jako demonstrační blok do této výpočetní dlaždice byl implementován filtr typu klouzavý průměr. Tento filtr byl zapojen do programovatelného DSP bloku, který umožňuje snadnější zpracování RTP paketu uloženého v generickém výpočetním bloku. Všechny tyto bloky prošly důkladnou verifikací a navíc jejich funkčnost byla ověřena připojením k procesoru OpenRISC, který tvoří jádro práce [5].

Do budoucna lze provést určitá vylepšení. Jako bezpodmínečně nutné vidím zprovoznění debugování skrze rozhraní JTAG. Ačkoliv toto rozhraní bylo zprovozněno v rámci práce [5], nepodařilo se mi jej oživit.

Jako další vylepšení by byla výhodná implementace DMA (*Direct Memory Access*). Obsluha hraničních bloků (*bridge slave* a *bridge master*) totiž procesor velmi zatěžuje. Řešení s DMA by mohlo pomoci k významnému snížení spotřeby (procesor by mohl být po dobu chodu DMA uspán). Pokud bude sytém BCI na FPGA implementován na RTL úrovni, bylo by výhodné dále optimalizovat programovatelný DSP blok.

10 Použitá literatura a zdroje

[1] **Chandrakasan, A.P.; Brodersen, R.W.,** "Minimizing power consumption in digital CMOS circuits," *Proceedings of the IEEE* , vol.83, no.4, pp.498,523, Apr 1995 doi: 10.1109/5.371964

[2] **Belhadj, Hichem , Aggrawal, Vishal, Pradhan, Ajay, Zerrouki, Amal,** "Power-aware FPGA design" **.** http://www.actel.com/documents/Power_Aware_WP.pdf

[3] **J. Šťastný**. "A Modular Hardware Platform for Brain Computer Interface," In Applied Electronics, Applied Electronics, pp 287-290, September 2012. ISBN 978-80-261-0038-6

[4] **J. Doležal, V. Černý, J. Šťastný**. "Constructing a Brain-Computer Interface," In Applied Electronics, Applied Electronics, pp 99-102, September 2011. ISBN 978-80-7043-987-6

[5] **P. Pavlata,** "Hardwarový akcelerátor pro BCI aplikace," Diplomová práce, České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra teorie obvodů, Praha, 2011

[6] **FPGA Laboratory,** Research Program Reports – Outcomes (30.3.2013) , http://amber.feld.cvut.cz/fpga/bci_resources/bci.wmv

[7] **OpenRISC**, "OR1200" Online (20.12.2013), http://openrisc.net/or1200-spec.html

[8] **J. Šťastný, J. Doležal,** "Real Time EEG Processing software" - system specification, nepublikovaná zpráva (30.3 2013)

[9] Osobní konzultace s autory BCI. České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra teorie obvodů, FPGA Laboratoř (2013).

[10] **J. Šťastný, J. Doležal, V. Černý, J. Kubový**. "Design of a modular brain-computer interface," In Applied Electronics, Applied Electronics, pp 319-322, September 2010. ISBN 978-80-7043-865-7

[11] **WISHBONE** System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores,Revision: B.3, Released: September 7, 2002. Online (20.12.2013), http://opencores.org/opencores,wishbone

[12] **Mohit Arora,** "The Art of Hardware Architecture – Design Methods and Techniques for Digital Circuits" , Springer 2012. ISBN 978-1-4614-0396-8

[13] **Šťastný Jakub,** „FPGA prakticky Realizace číslicových systémů pro programovatelná hradlová pole" , BEN – technická literatura, Praha 2010. ISBN 978-80-7300-261-9

[14] **WISHBONE** Builder. Online (20.12.2013), http://opencores.org/project,wb_builder

[15] UART 16550 core. Online (20.12.2013), http://opencores.org/project,uart16550

[16] Advanced Debug System. Online (20.12.2013), http://opencores.org/project,adv_debug_sys

[17] OpenRISC GNU tool chain. Online (20.12.2013),

http://opencores.org/or1k/OpenRISC_GNU_tool_chain

[18] **Lukáš Ručkay**, **Tomáš Kubec**, **Jakub Šťastný**. Návrh verifikačního prostředí pro FPGA bloky, Výzkumná zpráva #Z08-2. Online (20.12.2013), http://amber.feld.cvut.cz/fpga/publications/z08-2.pdf

11 Slovník zkratek

ASIC	Application-specific integrated circuit
BCI	Brain computer interface
CPU	Central processing unit
DDR	Double-data-rate synchronous Dynamic Random access memory
CMOS	Complementary Metal–Oxide–Semiconductor
DMA	Direct Memory Access
DSP	Digital signal processor
EEG	Elektroencefalogram
FPGA	Field programable gate array
FIR	Finite impulse response
FIFO	First In, First Out
GIB	General interface block
PC	Program Counter
PROM	Programmable Read Only Memory
RAM	Random-access memory
RISC	Reduced Instruction Set Computing
RTP	Real-time Transport Protocol
ROM	Read Only Memory
SPI	Serial Peripheral Interface
STA	Static Time Analysis
UDP	User datagram Protocol

12 Přílohy

Přehled přerušení v procesoru OpenRISC:

Exception Type	Vector Offset	SIGNAL	Example
Reset	0x100	None	Reset
Bus Error	0x200	SIGBUS	Unexisting physical location, bus parity error.
Data Page Fault	0x300	SIGSEGV	Unammaped data location or protection violation.
Instruction Page Fault	0x400	SIGSEGV	Unmapped instruction location or protection violation
Tick Timer Interrupt	0x500	None	Process scheduling
Alignment	0x600	SIGBUS	Unaligned data
Illegal Instruction	0x700	SIGILL	Illegal/unimplemented instruction
External Interrupt	0x800	None	Device has asserted an interrupt
D-TLB Miss	0x900	None	DTLB software reload needed
I-TLB Miss	0xA00	None	ITLB software reload needed
Range	0xB00	SIGSEGV	Arithmetic overflow
System Call	0xC00	None	Instruction l.sys
Trap	0xE00	SIGTRAP	Instruction l.trap or debug unit exception.

Přehled použití GPR registrů v procesoru OpenRISC :

Register	Preserved across function calls	Usage
R16	Yes	Callee-saved register
R15	No	Temporary register
R14	Yes	Callee-saved register
R13	No	Temporary register
R12	No	Temporary register (RVH - Return value high 32 bits of 64-bit value on 32-bit system)
R11	No	RV – Return value
R10	Yes	Callee-saved register
R9	Yes	LR – Link address register
R8	No	Function parameter number 5
R7	No	Function parameter number 4
R6	No	Function parameter number 3
R5	No	Function parameter number 2
R4	No	Function parameter number 1
R3	No	Function parameter number 0
R2	Yes	FP - Frame pointer
R1	Yes	SP - Stack pointer
R0	-	Fixed to zero

13 Obsah přiloženého CD

doc - dokument a vybraná dokumentace k blokům

firmware – zdrojové kódy testovacích aplikací pro OpenRISC, generátor arbitru sběrnice

OpenRISC – Projekt s připojenými bloky GIB k procesoru

src – HDL zdrojové kódy