Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

FIIT-13428-5813

Bc. Lukáš Kohútka

HARDVÉROVÁ AKCELERÁCIA OPERAČNÝCH SYSTÉMOV

Diplomová práca

Študijný program: Počítačové a komunikačné systémy a siete

Študijný odbor: 9.2.4 Počítačové inžinierstvo

Miesto vypracovania: Ústav počítačových systémov a sietí

Vedúci práce: Ing. Martin Vojtko

máj 2015

ANOTÁCIA

Slovenská technická univerzita v Bratislave

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný program: POČÍTAČOVÉ A KOMUNIKAČNÉ SYSTÉMY A SIETE

Autor: Bc. Lukáš Kohútka

Diplomová práca: Hardvérová akcelerácia operačných systémov

Vedúci diplomovej práce: Ing. Martin Vojtko

máj, 2015

V tejto práci sme navrhli hardvérovú akceleráciu plánovania úloh zameraného pre hard RT vnorené systémy. Aby sme dosiahli vysokú utilizáciu procesora a zvládli čo najviac úloh, rozhodli sme sa redukovať časovú zložitosť zvoleného plánovacieho algoritmu reálneho času použitím škálovateľnej hardvérovej architektúry založenej na paralelných posuvných registroch a prúdovom spracovaní inštrukcií. Náš plánovač úloh je navrhnutý ako koprocesor, ktorý implementuje algoritmus EDF. Vďaka dosiahnutiu konštantnej časovej zložitosti je plánovač nielen rýchlejší, ale taktiež aj deterministický. Ďalšou výhodou nášho plánovača je, že z plánovača môže byť odstránená príkazom ľubovoľná úloha, čo zvyšuje jeho rozšíriteľnosť a použiteľnosť v praxi. Pre overenie riešenia na platforme FPGA sme navrhli a použili špecifický BIST založený na funkcionálnom testovaní.

ANNOTATION

Slovak University of Technology Bratislava

FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGY

Degree Course: COMPUTER AND COMMUNICATION SYSTEMS AND NETWORKS

Author: Bc. Lukáš Kohútka

Master´s Thesis: Hardware Acceleration of Operating Systems

Supervisor: Ing. Martin Vojtko

2015, May

In this thesis, we have designed a hardware acceleration of task scheduling aimed for hard real-time embedded systems. In order to achieve high CPU utilisation and handle as many tasks as possible, we have decided to reduce time complexity of the chosen real-time scheduling algorithm by using scalable hardware architecture based on parallel shift registers and pipelining. Our task scheduler is designed as a coprocessor that implements EDF algorithm. Thanks to the achieved constant time complexity, the scheduler is not only faster but it is deterministic too. Another advantage of our task scheduler is that any task can be removed from the scheduler according to the task ID, which increases expandability and practical usefulness of the task scheduler. In order to verify our solution in FPGA, we have designed and used a specific BIST based on functional testing.

Čestné prehlásenie

Čestne prehlasujem, že som celú diplomovú prácu vypracoval samostatne pod odborným vedením vedúceho diplomovej práce, s využitím získaných poznatkov a na základe uvedených zdrojov.

V Bratislave, máj 2015 .....

Bc. Lukáš Kohútka

Poďakovanie

Poďakovanie je venované najmä vedúcemu tejto diplomovej práce, Ing. Martinovi Vojtkovi, ktorý bol nápomocný počas všetkých etáp riešenia práce formou poskytovania užitočných rád a upozorňovaním na potenciálne miesta vzniku problému. Taktiež by som rád poďakoval doc. RNDr. Elene Gramatovej, PhD. za to, že mi umožnila chodiť na semináre z digitálnych systémov, kde som mohol prezentovať výsledky mojej práce a získavať spätnú väzbu od zúčastnených učiteľov a doktorandov. Chcel by som tiež poďakovať Ing. Marošovi Ďuríčekovi za to, že ma naučil robiť syntézu na FPGA v programe Xilinx ISE. Za poskytnutie technických prostriedkov použitých na overenie riešenia na platforme FPGA ďakujem týmto ľuďom: doc. Ing. Tibor Krajčovič, PhD., Ing. Roman Záluský, PhD., Ing. Maroš Ďuríček a Ing. Martin Vojtko. Taktiež by som touto cestou rád poďakoval pánovi dekanovi doc. Ing. Pavlovi Čičákovi, PhD. a pani riaditeľke UPSS Ing. Kataríne Jelemenskej, PhD. za to, že mi umožnili zapísať si dva predmety zo študijného programu Mikroelektronika na FEI STU, ktoré súviseli s témou diplomovej práce. V neposlednom rade by som rád poďakoval mojej snúbenici Mirke, ktorá ma podporovala v práci na projekte.

Obsah

Zoznam obrázkov

Zoznam tabuliek

1 Úvod

Hardvérová akcelerácia je čoraz dostupnejšia a rozšírenejšia metóda, ako urýchliť existujúce algoritmy. Zatiaľ čo softvér je vo svojej podstate sekvenčné vykonávanie inštrukcií podľa napísaného počítačového programu, hardvér je schopný paralelného spracovania informácií s vysokou mierou efektivity. Jedným príkladom sú grafické karty, ktoré sa využívajú pri spracovaní obrazu alebo pri výpočtovo zložitých algoritmoch pomocou paralelizácie v podobe SIMD procesorov. Druhým príkladom sú sieťové karty podporujúce a urýchľujúce sieťovú komunikáciu medzi počítačmi.

Cieľom tejto práce je použiť programovateľné integrované obvody pre hardvérovú realizáciu aspoň jedného vybraného algoritmu vnoreného operačného systému, čím sa zníži réžia operačného systému a teda tým dosiahneme väčšie využitie procesora na plnenie úloh systému. Týmto môžeme zvýšiť výkonnosť celého počítačového systému, ktorý používa takúto hardvérovú akceleráciu operačného systému. Réžia operačného systému znižuje výkonnosť celého systému z toho dôvodu, že počas réžie operačného systému sa nevykonáva efektívny výpočet. Okrem toho platí, že väčšina teoretických plánovacích algoritmov predpokladá, že réžia operačného systému je nulová, čo znižuje predvídateľnosť a spoľahlivosť operačných systémov určených pre systémy reálneho času. Hardvérovo realizovaná časť operačného systému by mala vykonávať svoju funkcionalitu rýchlejšie a viac deterministicky v porovnaní so štandardným softvérovým riešením.

Nízka réžia operačného systému môže byť kriticky dôležitá pre vnorené systémy reálneho času, najmä hard RT vnorené systémy. Pre hard RT vnorené systémy je potrebné dodržiavať všetky stanovené limity pre odozvu systému, pretože len jedno nedodržanie môže spôsobiť zlyhanie celého systému, čo môže viesť k fatálnym následkom pre majetok, zdravie a život ľudí, ktorí sú súčasťou tohto systému. Napríklad oneskorená odozva výškomeru alebo autopilota v dopravných lietadlách môže spôsobiť haváriu lietadla a stratu na životoch. Preto najväčší prínos tejto práce vznikne vtedy, keď budeme hardvérovo akcelerovať taký operačný systém, ktorý sa využíva pre hard RT vnorené systémy.

Hardvérová akcelerácia operačného systému určeného pre hard RT vnorené systémy by mala redukovať pravdepodobnosť, že sa nejaká úloha nestihne dokončiť do stanoveného časového limitu určeného pre danú úlohu. Zmenšením réžie operačného systému a väčším využitím procesora na efektívny výpočet štatisticky zvyšujeme priemerný počet úloh, ktoré sa stihnú vykonať načas.

Okrem toho je hard RT systémy nesmierne dôležité, aby réžia operačného systému bola nielen malá, ale zároveň konštantne veľká. To znamená, že réžia OS by mala byť vždy

rovnaká bez ohľadu na akékoľvek parametre požitého algoritmu. Hardvérovou akceleráciou sa nám otvárajú dvere pre zložitejšie algoritmy, ktorých softvérová implementácia je nepraktická práve kvôli časovej zložitosti. Tieto zložitejšie algoritmy vedia lepšie rozhodovať o prideľovaní prostriedkov systému jednotlivým požiadavkám. V prípade plánovania úloh tým dosiahneme vhodnejšie poradie vykonania úloh, čo nám opäť zvyšuje priemerný počet úloh splnených načas.

1.1 Použité skratky a pojmy

V tomto dokumente boli použité viaceré skratky a pojmy. Zoznam skratiek a pojmov je nasledovný:

ALU	– Aritmeticko-logická jednotka (“Arithmetic Logic Unit”)
ASIC	– Aplikačne špecifický integrovaný obvod (“Application Specific Integrated Circuit”)
BIST	– Vstavané samočinné testovanie (“Built-In Self Testing”)
CPU	– Centrálna procesorová jednotka (“Central Processing Unit”)
Deadline	– maximálna povolená doba odozvy úlohy operačného systému
FPGA	– Programovateľný obvod (“Field Programmable Gate Array”)
FSL	– Komunikačné rozhranie pre hardvérovú akceleráciu (“Fast Simplex Link”)
ID	– Unikátny identifikátor
Koprocesor	– Hardvérový komponent, ktorý rozširuje funkcionalitu procesora
LED	– Svetlo emitujúca dióda (“Light Emitting Diode”)
LUT	– Vyhľadávacia tabuľka, ktorá je základnou bunkou v FPGA na realizovanie kombinačnej logiky (“Look-up Table”)
MOS	– Modulárny operačný systém
OS	– Operačný systém
Pipeline stage	– Stupeň prúdového spracovania inštrukcií
Pipelining	– Technika prúdového spracovania inštrukcií
RT OS	– Operačný systém reálneho času (“Real Time Operating System”)
Testbench	– Testovací obvod
VOS	– Vnorený operačný systém

1.2 Notácia UML

V tejto práci je použitá notácia UML, verzia 2.0, konkrétne pre diagram činností (“activity diagram”) a stavový diagram („state machine diagram“).

Začiatok diagramu:

Koniec diagramu:

Aktivita alebo stav (podľa druhu diagramu):

Rozhodovací blok:

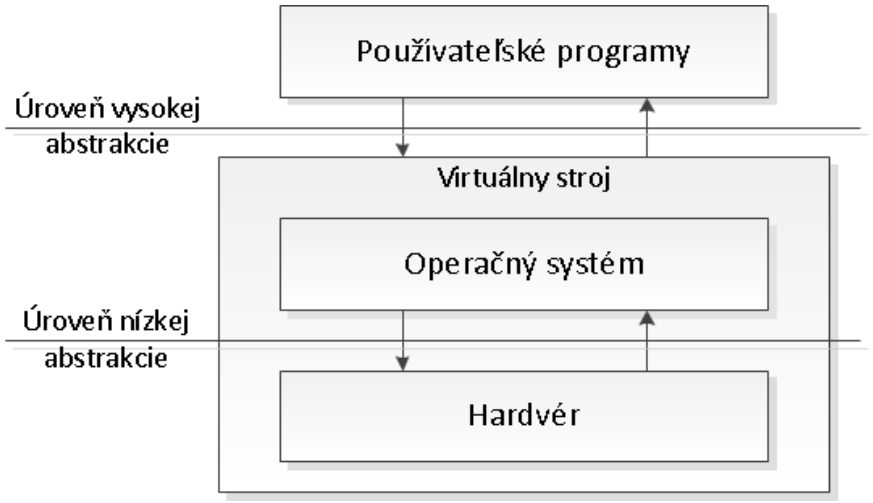
Riadiaci tok:

2 Analýza problému

V tejto kapitole sú definované dôležité pojmy, pravidlá a podobné riešenia potrebné pre úspešné vyriešenie problému vychádzajúceho zo zadania.

2.1 Pozícia a význam OS v počítačových systémoch

Operačný systém predstavuje rozhranie medzi hardvérom (procesorom, pamäťou a perifériami) a aplikačným softvérom (používateľské programy). Vďaka OS môžu vývojári aplikačného softvéru pracovať na vyššej úrovni abstrakcie, pretože neriešia záležitosti závislé od konkrétneho hardvéru a jeho prostriedkov. Tieto nízkoúrovňové záležitosti rieši operačný systém a tým vytvára z hardvéru virtuálny stroj (obr. 2.1). [VOJTKO, 2012]



Obr. 2.1: Operačný systém ako rozhranie medzi hardvérom a aplikáciami [VOJTKO, 2012]

Operačný systém môžeme teda chápať ako správcu prostriedkov. Jeho hlavnou úlohou je alokácia jednotlivých prostriedkov (procesorový čas, pamäť, periféria) aktívnym úlohám, ktoré tieto prostriedky potrebujú na svoje vykonanie. [TANENBAUM, 2001]

Operačný systém má časti platformovo závislé od konkrétneho hardvéru a časti platformovo nezávislé. Výhodou používania operačného systému je aj to, že používateľské programy nemusia byť tvorené výhradne pre špecifické hardvérové platformy (rodiny procesorov).

Nevýhodou operačného systému je skutočnosť, že sa jedná o softvérové rozšírenie hardvéru a teda volaním funkcií operačného systému sa systém spomaľuje. Počas vykonávania operačného systému vzniká tzv. réžia operačného systému. Tá predstavuje čas spotrebovaný vykonávaním operačného systému. Počas tohto času sa nevykonáva samotný kód aplikácie a teda vykonávanie aplikácie je spomaľované o réžiu operačného systému. Takáto réžia je v bežných počítačových systémoch (osobné počítače, laptopy, atď.) zanedbateľne malá. Avšak vnorené systémy (obzvlášť RT vnorené systémy) sú oveľa viac citlivé na výkon operačného systému, pretože nemajú k dispozícii menej prostriedkov a zároveň sú zvyčajne citlivejšie na dobu odozvy systému.

2.2 Delenie OS

Podľa oblasti pôsobenia delíme OS na [WOODHULL, 2006]:

- **OS pre bežných používateľov** – tieto operačné systémy sa vyskytujú najmä v osobných počítačoch a laptopoch. Sú určené pre bežných požívateľov, predpokladajú stredne veľkú výpočtovú silu, počet procesorových jadier a stredne veľkú pamäť. Majú bohaté GUI. Príkladom je Microsoft Windows a Apple MacOS.
- **Distribuované OS** – tieto operačné systémy sú zamerané pre väčšie výpočtové systémy známe ako distribuované počítačové systémy – počítačové gridy alebo klastre. Dôležitou vlastnosťou týchto systémov je veľké množstvo procesorových jadier (uzlov) a teda aj veľká miera paralelizmu.
- **OS pre vnorené systémy** – tieto operačné systémy sú naopak zamerané pre menšie výpočtové systémy, takže predpokladajú požitie menej výkonných procesorov a malého množstva pamäte. Vnorený systém je zväčša špecializovaný na určitú oblasť. Medzi VOS patria aj operačné systémy nasadené do mobilných telefónov, napríklad Android alebo Apple iOS. Avšak mnohé VOS ani nie sú určené pre manipuláciu používateľmi a teda ani nemusia mať GUI.

Podľa množstva používateľov a úloh delíme OS na [WOODHULL, 2006]:

- **Jedno-úlohový OS** – („single-user single-task“) operačný systém určený iba pre jedného používateľa v danom čase. Tento používateľ môže mať spustenú iba jednu úlohu. Výhodou takéhoto OS je predovšetkým jednoduchosť, pretože nie je potrebné riešiť prideľovanie úloh procesoru.
- **Viac-úlohový OS** – („single-user multi-task“) operačný systém použiteľný iba jedným používateľom v jednom čase, avšak je umožnené spracovanie viacerých úloh súčasne (paralelne alebo pseudoparalelne, čo závisí od počtu jadier procesora). Príkladom sú operačné systémy Microsoft Windows alebo Apple MacOS, v ktorých je prihlásený vždy práve jeden používateľ, ale tento používateľ bežne používa viacero aplikácií naraz.
- **Viac-používateľský OS –** („multi-user“) operačný systém, ktorý umožňuje prístup do systému viacerých používateľov súčasne. Dôležitou vlastnosťou takéhoto OS je snaha o rovnováhu prideľovania procesorového času jednotlivým používateľom tak, aby nespozorovali používanie systému ostatnými používateľmi. Príkladom takého systému je operačný systém Unix.

Podľa štruktúry delíme OS na [WOODHULL, 2006]:

- **Monolitický OS** – operačný systém, ktorý je implementovaný ako súvislý celok.
- **Kernel OS** – operačný systém, ktorý sa delí na jadro a ostatné nepovinné časti OS. Jadro obsahuje správu pamäte, správu úloh a správu vstupno-výstupného rozhrania.
- **Mikrokernel** – operačný systém, ktorého jadro obsahuje iba správu pamäte a správu úloh.
- **Nanokernel** – operačný systém, ktorého jadro obsahuje iba správu úloh.

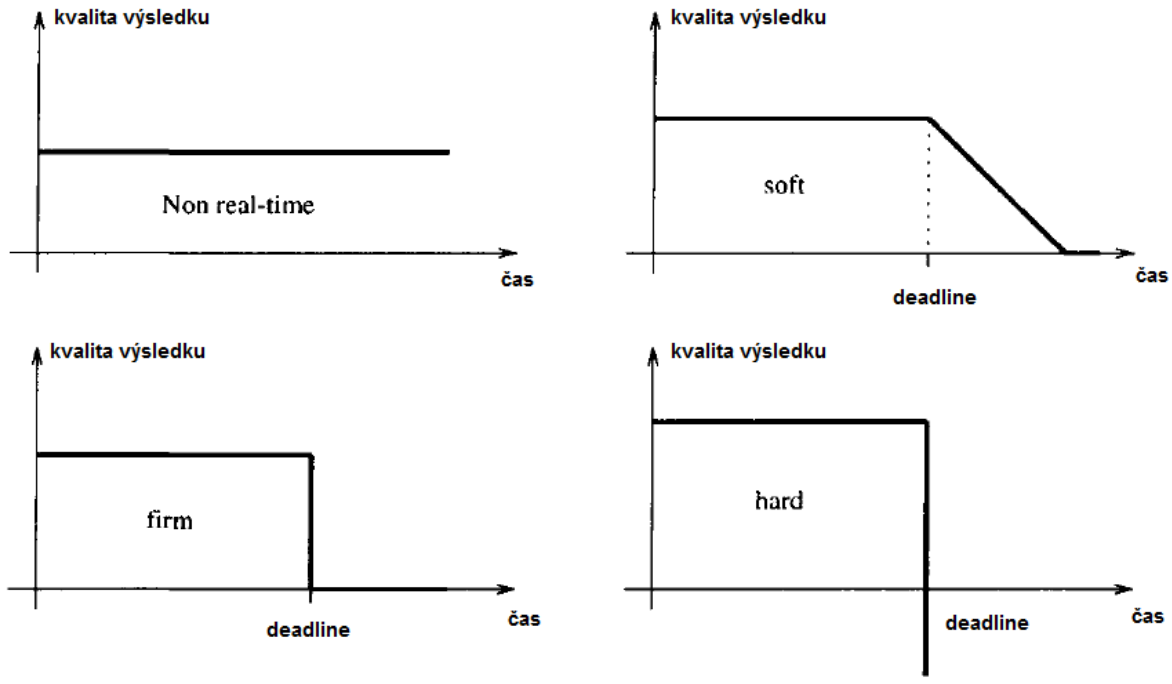
Podľa časovej odozvy delíme OS na [BUTTAZZO, 2011]:

- **RTOS** – operačný systém reálneho času, ktorý má definovanú maximálnu odozvu (deadline) pre jednotlivé úlohy. Tento OS sa vyskytuje zvyčajne vo vnorených systémoch, avšak nie je to pravidlom. Úlohy operačného systému reálneho času môžu byť rôzneho typu v závislosti od dôležitosti danej úlohy:
    - **Hard –** majú stanovený „deadline“, ktorého nedodržanie sa považuje za zlyhanie celého systému.

- **Firm** – majú stanovený „deadline“, ktorého nedodržanie síce spôsobí neplatnosť výsledku danej úlohy (ako keby táto úloha nebola vôbec vykonaná), avšak to nespôsobí zlyhanie celého systému.
- **Soft** – nemajú stanovený „deadline“ alebo prípadne majú, ale po jeho nedodržaní sa postupne znižuje kvalita výsledku úlohy.

- **Non real-time OS** – operačný systém bez záruky odozvy, vyskytujúci sa typicky v bežných osobných počítačoch.

Na nasledujúcom obrázku (obr. 2.2) môžeme vidieť rozdiely medzi úlohami z hľadiska doby odozvy. Po prekročení hranice „deadline“ klesá kvalita výsledku soft úlohy lineárne. V prípade firm úlohy klesne kvalita výsledku tejto úlohy ihneď na hodnotu 0, čiže akoby sa úloha ani nevykonala. V prípade hard úlohy klesne kvalita výsledku na $-\infty$. Celková kvalita systému je daná ako suma kvalít výsledkov jednotlivých úloh. Takže stačí, aby aspoň jedna hard úloha nedodržala stanovený deadline a celý systém má celkovú kvalitu $-\infty$. V prípade zriedkavého nedodržania soft úloh alebo firm úloh sa síce zhorší celková kvalita systému, avšak tento stav je stále akceptovateľný. [BUTTAZZO, 2011]



Obr. 2.2: Porovnanie úloh podľa časovej odozvy [BUTTAZZO, 2011]

2.3 Jadro OS a jeho časti

Jadro OS predstavuje základnú časť OS, ktorá je esenciálna pre každý operačný systém. Na rozdiel od ostatných častí je jadro OS vždy povinné pre použitie daného operačného systému. Nezáleží na tom, o ktorý typ OS sa jedná. Jadro OS sa označuje často pod pojmom kernel. Základnou úlohou každého jadra OS je spravovanie a prideľovanie prostriedkov jednotlivým úlohám. Každá úloha predstavuje požiadavku na vykonanie konkrétneho programu alebo jeho časti. Každý prostriedok je buď výpočtový alebo pamäťový. Keďže na splnenie úlohy (t.j. vykonanie programu) potrebujeme pamäť a procesor, tak pamäťové prostriedky reprezentujú pamäť alebo časť pamäte a za výpočtové prostriedky považujeme voľný čas procesora, ktorý môžeme použiť pre vykonanie programu. Okrem pamäte a procesorového času potrebujú úlohy zväčša aj komunikáciu s periférnymi zariadeniami, ktorá prebieha prostredníctvom vstupno-výstupného rozhrania [TANENBAUM, 2001]

Jadro OS pre vnorené systémy štandardne pozostáva z týchto častí [TANENBAUM, 2001]:

- správa úloh
- správa pamäte
- správa vstupno-výstupného rozhrania

Nasleduje popis jednotlivých častí OS a ich algoritmov.

2.3.1 Správa úloh

Najdôležitejšou úlohou jadra OS je správa úloh. V prípade minimalistického dizajnu OS (známeho tiež ako „nanokernel“) je jedinou časťou tohto dizajnu.

Podstatou správy úloh je prideľovanie procesorového času jednotlivým úlohám, ktoré o takýto procesor súperia. Procesor predstavuje prostriedok, ktorý potrebuje úloha, aby mohla byť vykonaná. Úloha je všeobecný výraz pre proces alebo vlákno. Proces je inštancia programu. Jeden proces môže pozostávať z viacerých vlákien, ale nie naopak. Rozdiel medzi procesom a vláknom je tiež v tom, že procesy navzájom nezdieľajú žiadne premenné, zatiaľ čo vlákna áno. Či už za úlohu je považovaný proces alebo vlákno, v každom prípade úloha predstavuje konkrétny kód, ktorý má procesor vykonať. [TANENBAUM, 2001]

Správa úloh pozostáva z týchto troch častí [LABROSSE, 2007]:

- **Prepínanie úloh** – predstavuje výmenu aktuálne vykonávanej úlohy inou úlohou. Najprv sa pozastaví doteraz vykonávaná úloha, následne sa odloží kontext tejto úlohy, načíta sa kontext novej úlohy a nakoniec procesor pokračuje vo vykonávaní novej úlohy. Kontext úlohy pozostáva minimálne zo všeobecných registrov, ktoré táto úloha používa a špeciálnych registrov ako je napríklad programové počítadlo (PC). Prepínanie úloh je zabezpečené hardvérovo priamo v procesoroch rodiny Intel x86. Ostatné procesory potrebujú softvérové prepínanie zabezpečené operačným systémom.
- **Komunikácia medzi úlohami –** je potrebná pre vzájomnú výmenu dát medzi úlohami v prípade, že tieto úlohy majú spolupracovať. Táto komunikácia môže byť zabezpečená rôznymi mechanizmami. Medzi najpoužívanejšie patrí posielanie správ, zdieľaná pamäť, synchronizácia (semafory) a vzdialené volanie procedúr.
- **Plánovanie úloh** – obsahuje vybraný plánovací algoritmus, podľa ktorého sa určuje poradie vykonávania jednotlivých úloh. Plánovanie úloh môže mať rôzne kritéria, podľa ktorých sa určuje poradie úloh v závislosti od zvoleného algoritmu. Výber algoritmu plánovania závisí aj od typu operačného systému, ktorý chceme vytvoriť. Napríklad RTOS používa iné algoritmy ako bežné OS.

Nasledujú najpoužívanejšie plánovacie algoritmy pre operačné systémy, ktoré nie sú reálneho času (Non real-time OS):

- **FIFO** - „First in, first out“ je algoritmus, ktorý ukladá úlohy do radu v takom poradí, v akom boli vytvorené. Tento algoritmus vykonáva úlohy ako celok, takže nasledujúca úloha sa začne vykonávať až po úplnom dokončení tej predchádzajúcej. Výhodou tohto algoritmu je minimálny počet prepínania úloh. Nevýhodou je to, že dlhšie vykonávané úlohy môžu príliš oneskoriť vykonanie kratších úloh.
- **Plánovanie s fixnými prioritami** – každá úloha musí mať pridelenú číselnú hodnotu, ktorá vyjadruje prioritu (dôležitosť) tejto úlohy. Tento algoritmus usporadúva úlohy podľa tejto priority. Nevýhodou tohto algoritmu je potreba manuálne zadávať každej úlohe jej prioritu podľa dôležitosti úlohy.
- **SJN** – „Shortest job next“ je algoritmus, ktorý usporadúva úlohy podľa dĺžky ich vykonania. Ide teda v princípe o plánovanie s fixnými prioritami, pričom priorita je vyjadrená dĺžkou vykonania danej úlohy a za predpokladu, že úlohy sú zoradené od najmenšej dĺžky po najväčšiu. Nevýhodou tohto algoritmu je potreba poznať dĺžky vykonávania úloh.

- **Round-robin –** každá úloha dostane rovnako veľký fixne daný diel procesorového času. Úlohy sa teda rovnomerne a opakovane striedajú vo vykonávaní. Je potrebné určiť vhodnú veľkosť časový interval pre výmenu úlohy, pretože príliš krátke intervaly spôsobujú príliš veľké množstvo prepínaní úloh a teda väčšiu réžiu OS. Na druhej strane, príliš dlhé intervaly zvyšujú oneskorenie, podobne ako v prípade algoritmu FIFO. Algoritmus Round-robin nepoužíva priority ani iné pomocné informácie o úlohách.

V praxi sa zvyknú úlohy rozdeliť do dvoch alebo viacerých skupín (napr. úlohy na popredí a úlohy na pozadí) a následne sa pre každú skupinu úloh použije jeden z vyššie spomínaných algoritmov. Úlohy na pozadí nepotrebujú takú rýchlu odozvu a teda je vhodné použiť FIFO alebo Round-robin s dlhšími časovými intervalmi. Úlohy na popredí potrebujú rýchlu odozvu a preto sa zvykne používať Round-robin s kratšími časovými intervalmi.

Operačné systémy reálneho času a taktiež vnorené operačné systémy potrebujú pre dosiahnutie časových hraníc „deadline“ používať iné algoritmy. Úlohy vnorených operačných systémov sa delia na:

- **Periodické** – periodicky sa opakuje vytvorenie takejto úlohy. Po uplynutí periódy sa automaticky vytvorí požiadavka na splnenie tejto úlohy od jej začiatku.
- **Aperiodické** – takéto úlohy môžu vznikať kedykoľvek, čiže v nepravidelných intervaloch.
- **Sporadické –** majú stanovený minimálny časový interval medzi vytváraním týchto úloh. Ak vznikne žiadosť o vytvorenie novej úlohy skôr, ako uplynie tento interval, tak takáto žiadosť sa odloží a úloha sa vytvorí až po uplynutí daného časového intervalu.

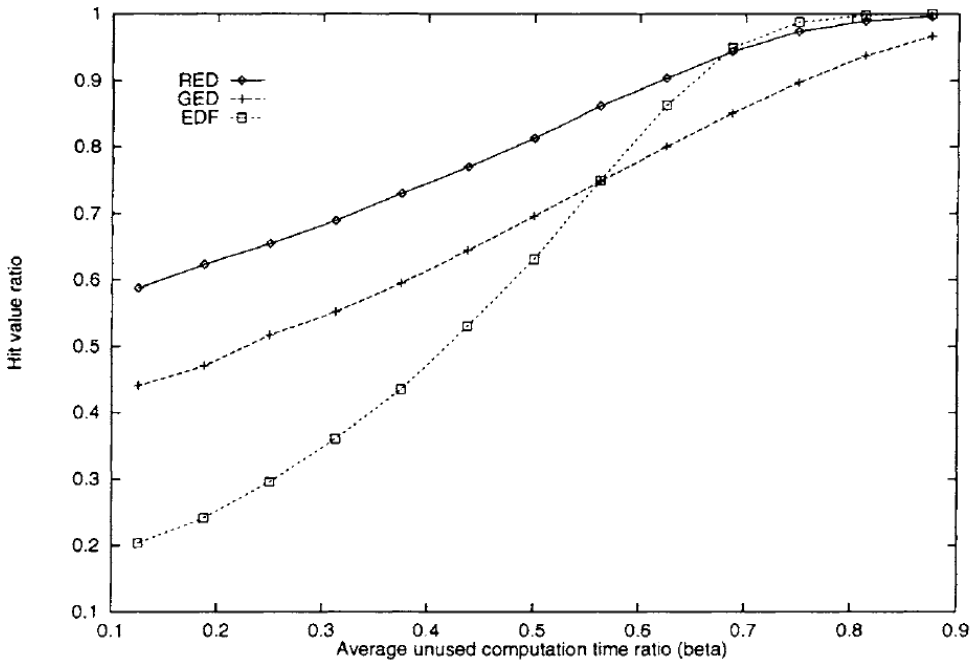
Najznámejšie plánovacie algoritmy operačných systémov určených pre systémy reálneho času sú:

- **RMS** – „Rate-monotonic scheduling“ je plánovací algoritmus, ktorý je na rozdiel od ďalších, tu vymenovaných, statický (ostatné sú dynamické). To znamená, že ešte pred zavedením systému do prevádzky sa pomocou tohto algoritmu určia priority jednotlivých úloh tak, aby boli splnené odozvy systému pre všetky tieto úlohy. Pre dynamické systémy tento algoritmus nie je vhodný a z hľadiska hardvérovej realizácie nie je vôbec zaujímavý.

- **EDF** – „Earliest deadline first“ je relatívne jednoduchý algoritmus, ktorý usporadúva úlohy podľa stanovených odoziev jednotlivých úloh. Čím skoršie má úloha stanovenú dobu odozvy, tým skôr sa úloha naplánuje. Tento algoritmus zaručuje splnenie všetkých dôb odoziev za predpokladu, že systém nie je preťažený. Preťaženie systému nastáva v prípade, že suma výpočtových časov, ktoré potrebujú úlohy na svoje splnenie, je vyššia ako procesorový čas, ktorý majú úlohy celkovo k dispozícii. V prípade, že táto podmienka nie je splnená a teda úloh je priveľa, neexistuje žiaden algoritmus, ktorý splní deadliny všetkých úloh. [BUTTAZZO, 2011]
- **LST** – „Least slack time“ narozdiel od algoritmu EDF zohľadňuje nielen deadline, ale aj čas, ktorý ešte potrebuje daná úloha na jej dokončenie. Rozdiel medzi deadlinom a časom potrebným na dokončenie úlohy sa nazýva „slack time“. Je to v podstate časová rezerva, ktorá zostáva úlohe na čakanie, kým sa nezačne vykonávať. Počas vykonávania iných úloh táto rezerva postupne klesá. V prípade, že „slack time“ klesne pod hodnotu 0, tak už úlohu nie je možné stihnúť dokončiť do stanoveného deadlinu. Algoritmus LST usporadúva úlohy podľa „slack time“ tak, že najprv sa vykonávajú úlohy s najnižším „slack time“. Tento algoritmus taktiež zaručuje splnenie všetkých deadlinov v prípade, že systém nie je preťažený. Nevýhodou LST oproti EDF je to, že algoritmus EDF potrebuje poznať iba deadliny, ale LST potrebuje poznať aj dĺžku vykonania jednotlivých úloh. [BUTTAZZO, 2011]
- **GED –** „Guarantee earliest deadline“ je modifikácia algoritmu EDF. Algoritmy EDF a LST sú typu „best effort“, pretože sa spoliehajú na to, že systém nebude preťažený. Algoritmus GED na rozdiel od EDF kontroluje, či pridanie novej úlohy do zoznamu naplánovaných úloh nespôsobí preťaženie systému. Preťažením systému môže nastať tzv. domino efekt, pretože nová úloha s relatívne krátkym deadlinom sa naplánuje pred ostatné úlohy, ktoré následne nedodržia svoj deadline. Domino efekty vzniká pri algoritmoch typu „best effort“. Algoritmus GED rieši tento problém nasledovným spôsobom. V prípade, že by pridanie tejto úlohy spôsobilo preťaženie systému, táto úloha sa nenaplánuje a jednoducho sa zahodí. Tento algoritmus je typu „guarantee“, pretože zaručuje, že keď sa raz úloha naplánuje, žiadna ďalšia nová úloha ju už neodstráni. Kontrola, či nastane preťaženie systému, sa realizuje tak, že sa priebežne sčítava celkový „slack time“ všetkých naplánovaných úloh. V prípade, že by táto hodnota klesla pod 0, systém bude preťažený. [BUTTAZZO, 2011]
- **RED –** „Robust earliest deadline“ je modifikácia algoritmu GED. Zatiaľ čo GED v prípade preťaženia systému danú úlohu zahodí, algoritmus RED túto úlohu aj tak naplánuje. Tento algoritmus radšej prejde zoznam všetkých naplánovaných úloh zoradených podľa deadlinu a vyberie za obeť takú úlohu, ktorá má najnižší parameter dôležitosti. Táto obeť sa úplne neodstráni, ale iba dočasne presunie do zoznamu odložených úloh s možnosťou opätovného naplánovania po znížení záťaže systému.

Tento algoritmus je „robust“, pretože je najzložitejší. Na druhej strane však dosahuje najlepšie výsledky. [BUTTAZZO, 2011]

Nasledovný obrázok (obr. 2.3) porovnáva účinnosť plánovacích algoritmov v prípadoch, keď je systém preťažený.



Obr. 2.3: Porovnanie účinnosti plánovacích algoritmov pre systémy reálneho času [BUTTAZZO, 2011]

Horizontálna os určuje mieru času, kedy systém nie je preťažený. Čím nižšia táto miera je, tým viac je systém preťažený. Vertikálna os vyjadruje mieru účinnosti plánovacieho algoritmu, čiže priemerný počet vykonaných úloh v stanovených časových ohraničeniach vzhľadom na celkový počet vytvorených úloh. Ako vidno na tomto obrázku, algoritmus EDF nie je vhodný v podmienkach preťaženého systému, pričom takéto podmienky v reálnom prostredí môžu nastať. Algoritmus RED je lepší ako GED, pretože RED úlohy hneď neodstraňuje, ale iba odloží s tým, že sa snaží o prečkanie preťaženého stavu a následné dokončenie odložených úloh. Celkovo teda možno zhrnúť, že najlepšie výsledky dosahuje algoritmus RED. Avšak jeho nevýhodou je, že jeho softvérová implementácia má príliš vysokú výpočtovú zložitosť.

Na základe vlastností jednotlivých plánovacích algoritmov určených pre systémy reálneho času môžeme zhrnúť, že:

- Ak systém obsahuje hard-RT úlohy, najvhodnejší algoritmus je EDF, pretože nemá zmysel sa zaoberať preťažením systému, nakoľko nie je dovolené odmietnuť naplánovať hard-RT úlohy.
- Ak systém obsahuje firm-RT úlohy, najvhodnejší algoritmus je GED, ktorý zabraňuje „domino efektom“ a teda väčšina úloh je splnená včas.
- Ak systém obsahuje soft-RT úlohy, najvhodnejší algoritmus je RED, ktorý používa primárny a sekundárny „deadline“.

2.3.2 Správa pamäte

Táto časť jadra OS implementuje dynamickú alokáciu pamäte a jej uvoľňovanie. Táto funkcionalita je známa v programovacom jazyku C ako dvojica funkcií:

- **Malloc** – ako vstup dostane veľkosť voľného bloku pamäte, ktorý má táto funkcia nájsť v hlavnej pamäti. Táto funkcia nájdený blok zmení z voľného na alokovaný (obsadený). Výstup funkcie je adresa začiatku tohto bloku pamäte.
- **Free –** ako vstup dostane adresu začiatku alokovaného (obsadeného) bloku pamäte. Táto funkcia daný blok zmení na voľný. Výstup nie je potrebný.

V rámci dynamickej alokácie môžu vznikať dva nepriaznivé javy, ktorým by sme sa mali čo najviac vyhýbať, pretože znižujú celkovú využiteľnosť pamäte. Tieto javy sú [TANENBAUM, 2001]:

- **Interná fragmentácia** – vzniká v prípade, keď je alokovaný väčší blok ako bola požadovaná veľkosť. Rozdiel medzi požadovanou veľkosťou úseku a reálne alokovanou veľkosťou sa navonok javí ako využívaná, avšak v skutočnosti je táto časť pamäte nevyužitá. Jedná sa teda o plytvanie pamäťou.
- **Externá fragmentácia** – vzniká v prípade, keď medzi jednotlivými alokovanými blokmi pamäte sa vyskytujú voľné úseky pamäte, ktoré sú natoľko malé, že sú prakticky len zriedka využiteľné. Tento problém vzniká v prípade striedavého alokovania a uvoľňovania rôznych veľkostí pamäte. Externú fragmentáciu je možné riešiť defragmentáciou (reorganizovanie alokovaných blokov pamäte).

Na alokáciu dynamickej pamäte potrebuje operačný systém prehľadať pamäť a nájsť dostatočné veľký voľný úsek pamäte. Na nájdenie tohto úseku existujú viaceré algoritmy [TANENBAUM, 2001]:

- **First-fit** – prehľadáva pamäť postupne od jej začiatku a prvý dostatočne veľký voľný blok pamäte použije. Výhodou je jednoduchá implementácia.
- **Next-fit** – začína prehľadávanie od naposledy alokovaného bloku pamäte a rovnako ako „first-fit“ použije prvý dostatočne veľký blok pamäte, ktorý nájde. Výhodou tohto algoritmu je, že vhodný blok nájde väčšinou pomerne rýchlo.
- **Best-fit** – vyberá najlepšie vyhovujúci voľný blok pamäte, čiže čo najmenší blok zo všetkých vyhovujúcich blokov pamäte. Tento algoritmus je pomalší ako predchádzajúce dva, pretože vždy potrebuje prehľadať celú pamäť a až potom vyberá blok, ktorý sa použije.
- **Worst-fit** – vyberá najhoršie vyhovujúci voľný blok pamäte, čiže ten, ktorý je najväčší. Tento algoritmus je paradoxne štatisticky lepší ako „best-fit“, pretože väčšina požadovaných blokov pamäte býva podobnej veľkosti.
- **Buddy algoritmus** – na rozdiel od predchádzajúcich algoritmov, tento algoritmus alokuje bloky v najmenšej väčšej veľkosti, ktorá je vždy mocninou čísla 2. Voľné úseky pamäte delí na polovice dovtedy, kým nedosiahne čo najmenší vyhovujúci úsek pamäte. Výhodou tohto algoritmu je, že úplne eliminuje externú fragmentáciu, ktorá vznikala v predchádzajúcich algoritmoch. Na druhej strane však vzniká interná fragmentácia, pretože tento algoritmus neprideľuje bloky presne požadovanej veľkosti, ale väčšie úseky tak, aby ich veľkosti boli mocniny čísla 2.

2.3.3 Správa vstupno-výstupných rozhraní

Táto časť jadra OS spravuje komunikáciu s vstupnými, výstupnými a vstupno-výstupnými zariadeniami, ktoré sú pripojené k danému počítačovému systému. Správa vstupno-výstupného rozhrania riadi prístup zariadení a výmenu dát medzi týmito zariadeniami a procesorom. Existujú tri typy komunikácie medzi procesorom a zariadením v závislosti od komunikačného protokolu, ktorý dané zariadenie používa [LABROSSE, 2007]:

- **Synchrónna** – táto komunikácia je riadená synchronizačným signálom z interných hodín procesora. Výhodou tohto prístupu je nižšia spotreba energie, pretože dané zariadenie sa môže vypínať a zapínať podľa synchronizačného impulzu. Nevýhodou je krátky dosah takejto komunikácie.

- **Asynchrónna** – táto komunikácia prebieha na báze výmeny synchronizačných znakov. Zariadenie neustále spracúva a vyhodnocuje dáta, ktoré prijíma a hľadá v nich synchronizačný znak. Takýto typ komunikácie si vyžaduje, aby zariadenie bolo nepretržite zapnuté.
- **Semi-synchrónna** – táto komunikácia je kombináciou synchrónnej a asynchrónnej komunikácie. Funguje na báze výmeny synchronizačných signálov. Zariadenie síce vie približne, kedy dôjde ku komunikácii, ale nevie to presne. Preto v danom časovom úseku je komunikácia synchronizovaná synchronizačným signálom. Tento prístup má výhody synchrónnej aj asynchrónnej komunikácie, pretože zariadenie sa môže vypínať, čím klesne spotreba energie, a zároveň nie je potrebné presné časovanie.

2.4 Ostatné časti OS

Okrem jadra operačného systému obsahuje OS zvyčajne aj ostatné časti, ktoré sú nepovinné. V niektorých prípadoch môžu byť niektoré časti priamou súčasťou jadra OS, ale môžu byť implementované ako samostatné moduly. Tieto moduly rozširujú jadro OS a tak spolu tvoria celý operačný systém. Medzi tieto ostatné časti OS patrí napríklad:

- správa súborového systému
- správa sieťového rozhrania
- terminál
- grafické používateľské rozhranie
- ovládače

Vyššie spomenuté časti nespadajú do zamerania tejto práce a preto sa nimi nebudeme hlbšie zaoberať.

2.5 Existujúce OS pre vnorené systémy

V tejto časti sú vymenované niektoré existujúce vnorené operačné systémy, na základe ktorých navrhneme model operačného systému, ktorým budeme simulovať fungovanie OS a jeho hardvérovú akceleráciu.

2.5.1 FreeRTOS

Tento operačný systém je najrozšírenejší zo všetkých OS zameraných pre systémy reálneho času. Je voľne dostupný aj pre komerčné použitie, čo z neho robí atraktívny operačný systém určený pre systémy reálneho času. Plánovanie úloh je zabezpečené prioritným plánovaním, pričom pre úlohy s rovnakou prioritou platí algoritmus Round Robin.

Nevýhodou tohto operačného systému je relatívne malé množstvo voľne dostupnej dokumentácie. Výhodou tohto OS je, že má relatívne jednoduchý zdrojový kód a podporuje širokú škálu procesorov používaných pre vnorené systémy (MicroBlaze, Nios II, PowerPC, ARM, x86 a mnohé ďalšie). [FREERTOS, 2013]

2.5.2 MOS

Ing. Martin Vojtko v rámci svojej diplomovej práce Modulárny operačný systém pre vnorené systémy navrhol a vytvoril MOS. Hlavnou výhodou tohto systému je modulárnosť, jednoduchosť, prehľadná dokumentácia a možnosť osobných konzultácií s autorom tohto systému. [VOJTKO, 2012]

Jadro tohto operačného systému je výlučne preemptívne a pozostáva zo správy pamäte, správy úloh a riadenia prístupu. MOS je obohatený o štatistický modul vďaka ktorému je možné odosielať štatistické dáta pre externú aplikáciu. K tomu bola implementovaná aplikácia, ktorá spracúva prichádzajúce štatistické dáta zo sériového rozhrania. [VOJTKO, 2012]

Tento operačný systém momentálne podporuje iba procesory ARMv4, čo je jeho asi najväčšou nevýhodou. Výhodou tohto systému v porovnaní s FreeRTOS je to, že MOS je pamäťovo efektívnejší a výrazne jednoduchšie rozšíriteľný o nové moduly.

2.5.3 MINIX 3

Tento operačný systém je podobne ako FreeRTOS založený na princípe „mikrokernel“. Väčšina zdrojového kódu tohto OS je vykonávané v používateľskom móde. Jeho najväčšou výhodou je, že tento systém je veľmi podrobne zdokumentovaný. Na plánovanie úloh používa 16 prioritných radov, ktoré fungujú podľa algoritmu Round-robin. Nevýhodou tohto OS je, že podporuje iba procesory x86 a ARM. [WOODHULL, 2006]

Aj tento systém je voľne dostupný pre komerčné použitie. Navyše na rozdiel od predchádzajúcich systémov ponúka MINIX 3 emulovateľné demo s terminálom.

2.5.4 NuttX

Tento OS je modifikáciou FreeBSD takým spôsobom, že podporuje systémy reálneho času. Má pomerne podrobnú a rozsiahlu dokumentáciu a taktiež je voľne dostupný aj pre komerčné účely. Medzi jeho hlavné črty patrí to, že je modulárny, škálovateľný a vysoko konfigurovateľný. Na plánovanie úloh používa FIFO a Round-robin algoritmus. [NUTTX, 2015]

Podporuje procesory ARM, Atmel AVR, x86 a Zilog.

2.5.5 Realtime Linux

Operačný systém Linux spolu s jeho rôznymi verziami je zo všetkých operačných systémov najrozšírenejší a najznámejší. Distribúcia Realtime Linux je modifikovaná tak, aby podporovala systémy reálneho času pomocou dodávaného rozšírenia (patch). Najväčšou výhodou tohto operačného systému, je že mnoho aplikácii je už vytvorených pre túto platformu. Nevýhodou je, že zdrojový kód nie je dostupný a stránka projektu venujúcemu sa tomuto OS má niekoľko rokov starý obsah. [RTWIKI, 2008]

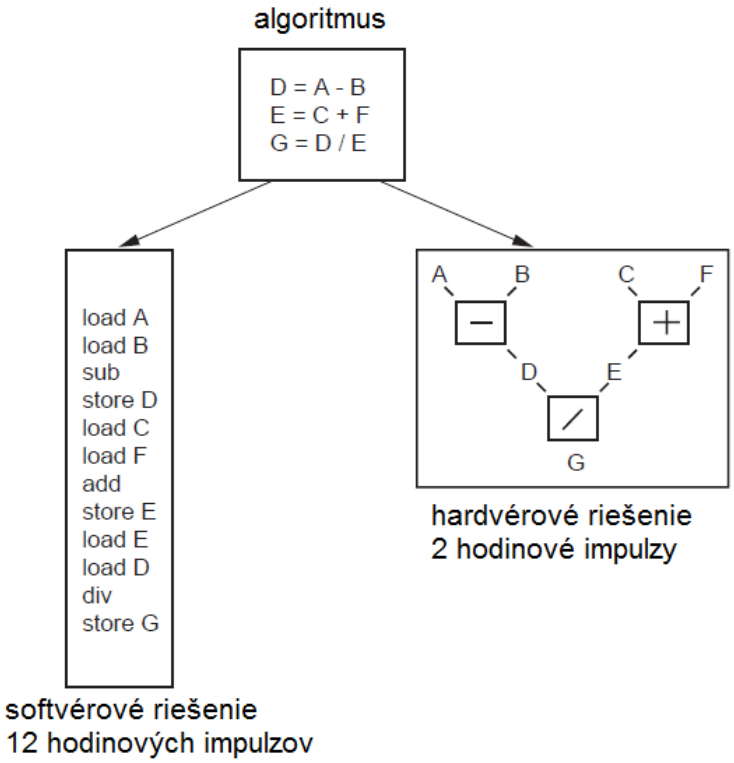
Tento OS podporuje procesory ARM, MIPS, PowerPC a x86.

2.6 Hardvérová akcelerácia a spôsoby realizácie

Hardvérový modul, ktorý urýchľuje vybraný výpočet, sa nazýva akcelerátor. Hardvérovú akceleráciu je možné dosiahnuť niektorým z nasledovných spôsobov, prípadne aj ich kombináciou [HENNESSY, 2011]:

- **Špecializovaný hardvér** – logické obvody realizované technológiou ASIC alebo FPGA. Procesory realizujú výpočty postupnosťou inštrukcií vykonávaných sekvenčne, takže výpočet si vyžaduje množstvo taktov procesora. Na rozdiel od toho špecializovaný hardvér môže realizovať celý výpočet inštrukciou špeciálne navrhnutou pre danú aplikáciu, ktorá potrebuje menej taktov ako procesor.
- **Dátový paralelizmus** – v prípade veľkého množstva dát, nad ktorými je nutné vykonať rovnaký výpočet. Hardvér môže dosahovať takú mieru paralelizmu nad dátami, akú daný výpočet potrebuje. Paralelizmus nad dátami môžeme dosiahnuť logickými obvodmi alebo architektúrou SIMD („Single-Instruction Multiple-Data“), ktorá je bežná v grafických procesoroch.
- **Procesný paralelizmus** – zatiaľ čo hardvérový akcelerátor realizuje daný výpočet, procesor môže vykonávať iný výpočet alebo inú úlohu. Takýto akcelerátor sa tiež nazýva koprocesor. Procesný paralelizmus možno dosiahnuť aj takým spôsobom, že zvýšime počet jadier procesora alebo pridáme ďalšie procesory, ktoré budú spolu tvoriť distribuovaný počítačový systém.

To, že hardvérové realizácie algoritmov sú vo všeobecnosti výkonnejšie ako softvérové, môžeme vidieť na nasledovnom príklade. Ako vidno z obrázku 2.4, hardvérové riešenie potrebuje na realizuje daného výpočtu 6-krát menej hodinových impulzov ako softvérové riešenie. Za predpokladu, že je v oboch prípadoch použitá rovnaká frekvencia synchronizačných hodín, je hardvérové riešenie 6-krát rýchlejšie ako softvérové. Miera zníženia počtu synchronizačných impulzov hardvérovou realizáciou algoritmu závisí od konkrétneho algoritmu a možností jeho paralelizácie.



Obr. 2.4: Porovnanie softvérového riešenia s hardvérovým [ROSINGER, 2004]

Na realizáciu prototypu hardvérovej akcelerácie je ideálne použiť konfigurovateľné obvody známe ako FPGA, pretože v malom množstve sú výrazne lacnejšie a dajú sa kedykoľvek preprogramovať. Pre masovú výrobu je výhodnejšie použiť technológiu ASIC, ktorá má síce svoju štruktúru fixne danú, ale optimalizovanú. [GOKHALE, 2010]

Komunikácia medzi hardvérovým akcelerátorom a univerzálnym procesorom, ktorého operačný systém má byť akcelerovaný, je realizovateľná viacerými spôsobmi [GOKHALE, 2010]:

- **Prostredníctvom systémovej zbernice** – tento prístup by bol relatívne jednoduchý pre používateľa, avšak komunikácia by mala príliš veľké oneskorenie a tým by sa oslabila samotná akcelerácia daného výpočtu.
- **Pripojením akcelerátora na adresnú a dátovú zbernicu procesora –** v tomto prípade nie je problém s oneskorením spôsobeným prenosom údajov medzi CPU a koprocesorom, keďže táto komunikácia môže trvať iba jeden takt procesora. Problém však vzniká vtedy, keď máme viacero prvkov pripojených na túto spoločnú

zbernicu, ktoré môžu žiadať o použitie zbernice v rovnakom čase. Tento problém narastá v moderných počítačových systémoch, kde sa používajú CPU s viacerými jadrami.

- **Ako koprocesor –** tento prístup je podmienený tým, aby bol akcelerátor integrovaný priamo na čipe s CPU a rozširoval inštrukčnú sadu daného procesora. Výhodou tohto riešenia je, že komunikácia má minimálnu latenciu a zároveň je nezávislá od adresnej a dátovej zbernice. Preto je takéto riešenie najvhodnejšie. Treba však dbať na to, aby dĺžka kritickej cesty koprocesora bola menšia alebo rovná dĺžke kritickej cesty CPU, inak by sme znižovali výkon systému. [STEPHENSON, 2005]

Existujú 3 typy procesorov, ktoré môžeme mať v systéme na čipe alebo v zariadení FPGA [LYSECKY, 2005]:

- **Soft core** – procesor je opísaný špecifikačným jazykom pre hardvér (napr. VHDL) na úrovni RTL. Tento opis sa prekladá a konfiguruje priamo do logiky FPGA. Výhodou je, že autor si môže navrhnúť vlastný procesor a upraviť jeho rozhranie pre akcelerátor. Celý systém môže byť synchronizovaný jednými hodinami s tou istou frekvenciou. Navyše takýto opis je možné ľahko použiť pre rôzne platformy. Nevýhodou je, že takáto realizácia je prostredníctvom technológie FPGA pomerne neefektívna z hľadiska spotrebovanej plochy na čipe a rýchlosti procesora. Takéto riešenie je ideálne na vytvorenie prototypu systému a jeho otestovanie. Následne môže návrh celého systému poslúžiť pre výrobu ASIC obvodov.
- **Firm core** – podobne ako v prípade „soft core“, aj tento procesor je opísaný pomocou jazyka VHDL alebo podobného jazyka. Rozdiel je v tom, že takýto procesor si nemôže návrhár navrhnúť sám, pretože v tomto prípade ide o IP jadro, ktoré je optimalizované pre jednu konkrétnu platformu a nie je možné meniť štruktúru tohto jadra. Príkladom je procesor MicroBlaze od firmy Xilinx a Nios II od firmy Altera.
- **Hard core** – procesor je fyzicky realizovaný na čipe v technológii s pevne danou logikou (podobne ako ASIC). Výhodou je optimálna štruktúra a teda aj vyššia taktovacia frekvencia procesora. Nevýhodou je skutočnosť, že takýto procesor pracuje rýchlejšie ako obvody realizované pomocou FPGA, čím je nutné zabezpečiť synchronizáciu komunikácie medzi procesorom a FPGA. Medzi hard core procesory patrí napríklad IBM PowerPC alebo ARM.

Pre potreby samotnej verifikácie a testovania hardvérového akcelerátora je možné nahradiť procesor a OS testovacím prostredím, napríklad BIST.

2.7 Existujúce hardvérové riešenia

Táto podkapitola sa delí na dve časti. V prvej časti je analyzovaný najdôležitejší komponent systému, ktorého cieľom je zoraďovať úlohy operačného systému podľa určitého kritéria. Druhá časť obsahuje vybrané existujúce riešenia hardvérového plánovača úloh pre systémy reálneho času.

2.7.1 Hardvérová realizácia zoraďovania položiek

Väčšina plánovacích algoritmov potrebuje úlohy zoraďovať podľa nejakého kritéria, najmä ak ide o vnorené operačné systémy reálneho času. Napríklad algoritmus EDF potrebuje úlohy zoraďovať podľa hodnoty „deadline“ a algoritmus LST zoraďuje úlohy podľa hodnoty „slack time“. Preto sa budeme v ďalšej časti venovať existujúcim spôsobom hardvérovej realizácie zoraďovania položiek.

Medzi najznámejšie architektúry patrí [CHANDRA, 2010]:

- **Zoznam FIFO štruktúr** – systém obsahuje toľko FIFO štruktúr, koľko priorít chceme podporovať. Nová položka sa vkladá do tej FIFO štruktúry, ktorá prináleží priorite danej položky. Toto riešenie je veľmi zle škálovateľné z hľadiska počtu priorít ako aj z hľadiska počtu položiek. [MOON, 2000]
- **MUX stromy** – položky sa načítavajú sériovo do FIFO štruktúry, pričom položka s maximálnou alebo minimálnou hodnotou sa určuje pomocou úplnej stromovej štruktúry, ktorej každý uzol predstavuje multiplexor a komparátor. Takéto riešenie je najhoršie škálovateľné z hľadiska plochy na čipe, ako aj z hľadiska rýchlosti systému vyjadreného frekvenciou systému, ktorá závisí od dĺžky kritickej cesty.
- **Posuvné registre** – položky sa načítavajú paralelne do všetkých N lineárne zapojených buniek, pričom každá bunka obsahuje komparátor. Všetky bunky majú ľavého a pravého suseda a navzájom si vymieňajú informácie o tom, v akom stave sa nachádzajú. Zároveň je možné posúvať položku, čiže obsah bunky z jednej bunky do susednej. Nová položka sa vloží na miesto existujúcej položky, ktorá sa presunie do jednej strany spolu so zvyškom položiek v danom smere. Toto riešenie je optimálne z hľadiska plochy na čipe, avšak kvôli paralelnému čítaniu novej položky do všetkých buniek sa lineárne predlžuje šírenie signálu po zbernici v závislosti od N a teda aj kritická cesta.

- **Systolické pole** – štruktúra veľmi podobná posuvným registrom, ale s tým rozdielom, že položky sa načítavajú sériovo z toho istého konca, odkiaľ získavame aj výstup z obvodu. Tento jav je dosiahnutý zdvojnásobením počtu buniek, pričom kapacita zostáva pôvodná. Takéto riešenie je optimálne z hľadiska rýchlosti, keďže zvyšovanie počtu buniek neovplyvňuje oneskorenie obvodu. Na druhej strane je toto riešenie neefektívne z hľadiska plochy na čipe, pretože vyžaduje 2 *N* buniek.
- **Kombinácia posuvných registrov a systolického poľa** – štruktúra identická so systolickým poľom len s tým rozdielom, že každá bunka tohto poľa predstavuje posuvné registre a až tie následne obsahujú *M* položiek, ktoré sa zoraďujú a teda *M* + 1 buniek. Takýto prístup kombinuje výhody oboch pôvodných architektúr. Celkový počet buniek potrebných pre túto architektúru je

$$N + \frac{N}{M} \tag{1}$$

  Čím väčšie je *M*, tým menej buniek potrebujeme, avšak na druhej strane tým väčšie je oneskorenie spôsobené šírením signálu v posuvných registroch. Oneskorenie nezávisí od *N* (rovnako ako u štandardného systolického poľa), ale od *M*. [MOON, 2000]

2.7.2 Hardvérová realizácia plánovača úloh

Existuje veľké množstvo hardvérových plánovačov úloh a preto boli vybrané a opísané len týchto 5 riešení vymenovaných podľa mien ich autorov, keďže niektoré z nich nedostali od svojich autorov žiadne pomenovanie:

- **KOHOUT** – realizuje iba plánovací algoritmus RMS, pričom úlohy zoraďuje pomocou MUX stromu. Kapacita úloh je 64. Viac informácii o architektúre systému nebolo zverejnených. [KOHOUT, 2002]
- **BLOOM** – toto riešenie realizuje algoritmy RMS a EDF. Na zoraďovanie úloh používa pole FIFO štruktúr. Viac informácii o architektúre systému nebolo zverejnených. [BLOOM, 2010]
- **FERREIRA** – operačný systém s názvom OReK. Spolupracuje s hard core procesorom IBM PowerPC 405. Plánovanie je buď statické (RMS, DMS) alebo dynamické (EDF). Úlohy so zoraďované pomocou MUX stromu. Maximálny podporovaný počet úloh je iba 32. [FERREIRA, 2010]
- **LANGE** – toto riešenie má názov HartOS. Hardvérový modul spolupracuje so soft core procesorom Xilinx MicroBlaze pomocou rozhrania FSL. Realizovaný je iba

plánovací algoritmus RMS, pričom na zoraďovanie úloh sa používa MUX strom. [LANGE, 2012]

- **ONG** – tento operačný systém sa volá SEOS. Podporované je iba statické plánovanie pomocou algoritmu RMS, pričom maximálny počet úloh je 64. Podrobnosti o architektúre nie sú zverejnené, dokonca ani spôsob zoraďovania úloh. [ONG, 2013]

2.8 Zhodnotenie analýzy

V tejto kapitole sme analyzovali jadro vnorených operačných systémov a jeho časti, algoritmy pre správu úloh a správu pamäte, plánovanie úloh pre systémy reálneho času, možnosti hardvérovej akcelerácie algoritmov, existujúce hardvérové riešenia plánovania úloh a hardvérové zoraďovanie položiek, ktoré je esenciálne pre plánovanie úloh operačného systému. Môžeme zhodnotiť, že existuje široké spektrum možností v každej časti analýzy. Cieľom tejto analýzy bolo oboznámiť sa s problematikou danej témy, aby sme boli schopní použiť niektoré existujúce metódy a postupy pre návrh nového systému.

Z oblasti algoritmov správy úloh využijeme poznatky o existujúcich algoritmoch plánovania úloh pre systémy reálneho času. Na základe tejto analýzy sme si vybrali algoritmus EDF, ktorý je ideálny pre hard-RT systémy reálneho času.

Z oblasti existujúcich spôsobov hardvérovej akcelerácie boli vybrané 2 FPGA, jedno od firmy Xilinx a druhé od firmy Altera. Akcelerátor bude navrhovaný vo forme koprocesora, ktorého architektúra ako aj rozhranie bude univerzálne použiteľné pre ľubovoľné CPU a OS.

Z oblasti existujúcich operačných systémov nebol vybraný žiaden existujúci OS. Je to z toho dôvodu, že považujeme za jednoduchšie a efektívnejšie navrhnúť vlastné testovacie prostredie, ktoré bude simulovať správanie ľubovoľného operačného systému na ľubovoľnom CPU. Takéto testovacie prostredie bude generovať inštrukcie pre koprocesor, ktorý predstavuje hardvérový plánovač úloh určený pre hard RT vnorené systémy. Testovacie prostredie by malo byť založené na funkcionálnom testovaní a metóde BIST, pričom by mali byť tieto dve metódy skombinované do jednej, ktorú budeme nazývať funkcionálny BIST.

Z oblasti existujúcich spôsobov hardvérovej realizácie zoraďovania položiek bolo vybrané riešenie, ktoré kombinuje paralelné posuvné registre s prúdovým spracovaním inštrukcií. Toto riešenie by malo byť čo najlepšie škálovateľné ako z hľadiska rýchlosti, tak aj z hľadiska spotreby plochy na čipe. Takáto architektúra je výhodnejšia ako všetky existujúce alternatívy.

Napriek veľkému množstvu existujúcich riešení hardvérového plánovania úloh je potrebné upozorniť na nasledovné problémy:

- Nebolo nájdené žiadne existujúce riešenie, ktoré by bolo akokoľvek nasadené do praxe alebo aspoň všeobecne uznané ako štandard.
- Skoro všetky nájdené riešenia používajú iba tie najjednoduchšie plánovacie algoritmy, prevažne statický algoritmus RMS alebo plánovanie úloh založené na priorite úloh.
- Všetky nájdené riešenia používajú veľmi zle škálovateľné a teda neefektívne spôsoby zoraďovania úloh, ktoré limitujú počet úloh v systéme.
- Architektúra systému je málo opísaná, niekedy dokonca chýbajú základné informácie o danom riešení a preto bude takmer nemožné porovnať naše riešenie s existujúcimi.
- Nie je jednotný spôsob testovania funkčnosti a kvantitatívnych vlastností navrhovaných riešení. Aj preto sa výsledky testovania ani nedajú, prípadne dajú len veľmi nepresne porovnať s inými výsledkami.

3 Opis riešenia

Táto kapitola obsahuje postup riešenia, ktorý pozostáva zo špecifikácie požiadaviek jednotlivých častí systému, návrhu jednotlivých častí smerom zhora nadol, ktoré sú opísané textovo, opisnými jazykmi ako je napríklad VHDL a grafickým opisom, čiže schémami zapojenia štruktúry. Nechýbajú ani opisy algoritmov pomocou UML diagramov a pseudokódov. V podkapitole *špecifikácia požiadaviek* je zadefinované, čo potrebujeme urobiť pre splnenie zadania, zatiaľ čo podkapitola *návrh riešenia* obsahuje postup riešenia.

3.1 Špecifikácia požiadaviek

Cieľom práce je navrhnúť a vytvoriť hardvérový plánovač úloh vo forme koprocesora, ktorý bude vedieť spolupracovať s ľubovoľným CPU a OS.

3.1.1 Funkcionálne požiadavky

Navrhovaný koprocesor by mal realizovať vhodný plánovací algoritmus využívajúci dobu odozvy úloh, pričom zároveň umožňuje odplánovať ktorúkoľvek úlohu, nie iba úlohu na začiatku zoznamu úloh. Preto je požiadavkou, aby koprocesor podporoval minimálne 2 inštrukcie:

- *task_schedule* – vloží novú úlohu do koprocesora a naplánuje túto úlohu na základe jej maximálnej doby odozvy (*deadline*).
- *task_kill* – odstráni ľubovoľnú úlohu z koprocesora na základe ID úlohy.

3.1.2 Nefunkcionálne požiadavky

Základnou požiadavkou na každý systém je to, aby mal čo najväčší prínos (pridanú hodnotu), najmä ak sa jedná o návrh hardvéru. V našom prípade chceme aplikovať hardvérovú akceleráciu na zníženie réžie operačného systému – hardvérový plánovač úloh. Nízka réžia operačného systému je obzvlášť dôležitá pre systémy reálneho času. Preto by mal navrhovaný operačný systém podporovať hard-RT úlohy. Táto požiadavka je však automaticky splnená funkcionálnou požiadavkou na realizáciu algoritmu EDF.

Dôležitou požiadavkou na navrhovaný systém je aj to, aby bolo možné splniť čo najviac úloh do stanovených časových ohraničení a teda aby bol systém používajúci tento operačný systém čo najkvalitnejší. Túto požiadavku je možné vyjadriť dvomi cieľmi:

1) Aby bol plánovací algoritmus čo najrozumnejší a teda aby sa robilo čo najmenej nesprávnych rozhodnutí plánovania úloh. Je zbytočné mať veľkú výpočtovú silu, keď sa nevyužíva efektívne.
2) Aby prebiehalo plánovanie čo najrýchlejšie a teda aby bolo čo najviac výpočtového času spotrebovaného jednotlivými úlohami, nie operačným systémom. Veľká réžia operačného systému znižuje výkonnosť celého systému. Ideálne by bolo, keby malo plánovanie konštantnú časovú zložitosť, aby bolo oneskorenie spôsobené prepínaním úloh deterministické.

Vyššie spomínané dve požiadavky sú protichodné, pretože znižovaním zložitosti plánovacieho algoritmu dosahujeme rýchlejší, avšak menej rozumný algoritmus. Naopak zložitejšie algoritmy sú rozumnejšie, avšak zároveň pomalšie. Tento problém je možné riešiť práve hardvérovou akceleráciou, kedy použijeme zložitejší algoritmus, ktorý dáva rozumnejšie poradie úloh a zároveň je rýchly a deterministický.

Keďže sa jedná zároveň o digitálny systém, dôležitými požiadavky na tento hardvér sú:

- Relatívne malé množstvo hardvérových prostriedkov – pri ASIC ide o plochu na čipe, pri FPGA ide o počet LUT, počet prepojení a množstvo pamäte.
- Relatívne nízky príkon – obzvlášť dôležitý parameter pre mobilné zariadenia. Aj v prípade, keď sa nejedná o mobilné zariadenie, je tento parameter dôležitý, pretože nižší príkon spôsobuje aj [HENNESSY, 2011]:
  - nižšia spotreba energie – nižšie prevádzkové náklady
  - nižšia výhrevnosť systému – čoraz viac obmedzujúce pri zmenšovaní tranzistorov
  - vyššia životnosť a teda aj spoľahlivosť systému
- Relatívne vysoká rýchlosť systému – vysoká frekvencia, nízke oneskorenie jednej odpovede a vysoká priepustnosť systému v prípade viac požiadaviek na plánovač úloh za sebou. Ideálna doba odozvy je konštantná, čiže nezávisí od počtu úloh v plánovači.
- Podpora čo najviac úloh – závisí priamo od kapacity plánovača a nepriamo od jeho rýchlosti. Je zbytočné mať extrémne veľkú kapacitu plánovača, keď jeho rýchlosť aj tak neumožňuje stihnúť naplánovať a následne splniť ani polovicu úloh z kapacity plánovača. Opačný extrém by bol rovnako málo užitočný.

- Kompatibilita systému – aby bolo možné jednoducho rozšíriť existujúce procesory navrhovaným hardvérovým modulom. Na to je potrebné, aby mal navrhnutý hardvérový plánovač komunikačné rozhranie nezávislé od architektúry procesora.

Spotreba plochy čipu je protichodná požiadavka voči rýchlosti systému a preto by mal návrh systému obsahovať určitý kompromis medzi vyššie spomínanými požiadavkami.

3.2 Návrh riešenia

Nasleduje opis návrhu riešenia, ktoré pozostáva z nového plánovacieho algoritmu, návrhu systému na najvyššej (tzv. systémovej) úrovni a opis komponentov, ktoré predstavujú jednotlivé časti systému.

3.2.1 Nový plánovací algoritmus FED

Nový plánovací algoritmus sa volá FED. Tento názov je odvodený od slov „Flexible Earliest Deadline“, ktoré znamenajú flexibilný najskorší čas odozvy. Algoritmus FED je inšpirovaný od algoritmu RED a preto má aj podobný názov. Tento algoritmus patrí do skupiny robustných plánovacích algoritmov pre systémy reálneho času. To znamená, že rovnako ako RED, aj FED kontroluje, či nenastáva preťaženie systému a tiež používa zoznam odložených úloh pre prípadne opätovné naplánovanie úlohy. Zásadný rozdiel medzi algoritmami FED, RED a EDF je ten, že zatiaľ čo RED je určený iba pre soft RT úlohy a EDF je určený iba pre hard RT úlohy, tak algoritmus FED podporuje systémy, ktoré ľubovoľne kombinujú používanie úloh typu hard RT, firm RT a non-RT. Túto vlastnosť sme získali tým, že na rozdiel od algoritmu RED, ktorý používa dva druhy časovej odozvy (primárnu a sekundárnu), tak FED používa iba jeden druh časovej odozvy. Na algoritmus FED sa môžeme tiež pozerať ako iba na modifikáciu algoritmu RED, v ktorom sme pridali podmienku:

$$\text{primárna časová odozva} = \text{sekundárna časová odozva} \qquad (2)$$

Zároveň však treba dodať, že algoritmus FED má aj presne špecifikovaný postup výberu úlohy, ktorá sa má vyradiť zo zoznamu naplánovaných úloh v dôsledku preťaženia systému. Algoritmus RED tento postup neobsahuje a necháva túto problematiku na inžinieroch, ktorí chcú RED použiť. Ďalšou výhodou algoritmu FED oproti RED je fakt, že FED nepotrebuje požívať žiadne delenie ani násobenie vo svojich výpočtoch, zatiaľ čo RED áno. Vďaka tomu

je FED vhodnejší pre hardvérové realizácie, keďže násobenie a delenie zvyčajne neprebiehajú v rámci jedného cyklu synchronizačných hodín.

Na fungovanie algoritmu FED potrebujeme viaceré údaje o úlohách. Niektoré z nich potrebujeme získať externe a niektoré môžeme vypočítať na základe ostatných informácií o úlohách. Údaje potrebné pre algoritmus FED sú:

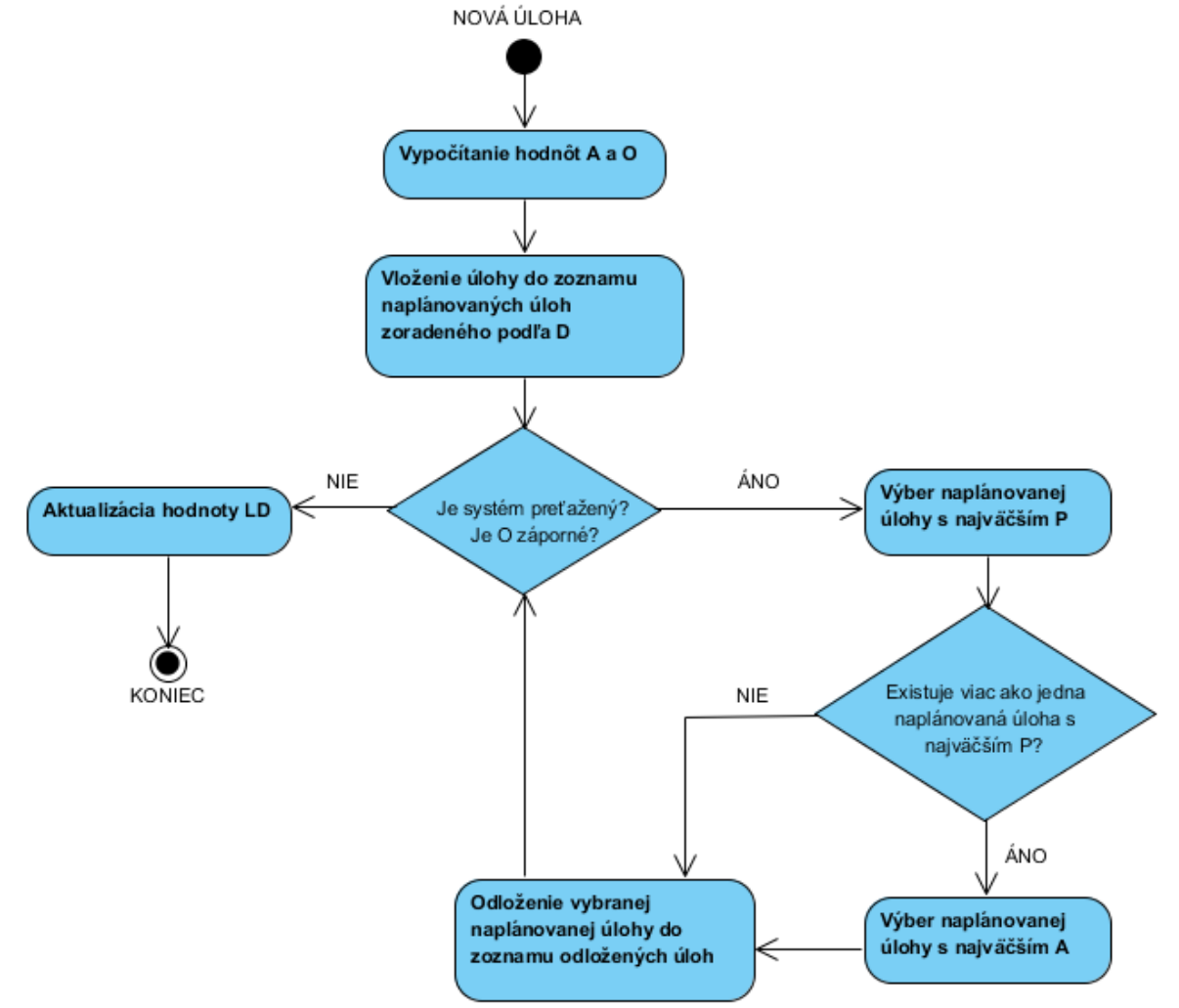
- ***D*** – relatívna doba odozvy úlohy, čiže časový úsek od aktuálneho času až po maximálny čas, dokedy je potrebné danú úlohu splniť. Podľa tejto hodnoty sú zoradené naplánované úlohy. Okrem relatívnej doby odozvy úlohy môžeme v prípade potreby použiť iný čas:
  - absolútna doba odozvy úlohy – časová pečiatka (t. j. presný dátum a čas).
  - Prírastková doba odozvy úlohy – časový úsek od absolútnej doby odozvy predchádzajúcej úlohy po absolútnu dobu odozvy nasledujúcej úlohy. V prípade, že sa jedná o prvú úlohu, tak je táto hodnota rovnaká ako relatívna doba odozvy úlohy.
- ***C*** – najhorší možný čas vykonania danej úlohy. Existujú dve základné možnosti ako tento čas zistíme:
  - analyticky – vypočítame podľa zdrojového kódu danej úlohy.
  - empiricky – na základe predchádzajúcich vykonaní úloh a ich časov splnenia je nová hodnota *C* rovná maximálnej hodnote z predchádzajúcich výsledkov. Počiatočná hodnota *C* pre prvé spustenie úlohy je 0.
  - kombinácia – počiatočná hodnota *C* je určená na základe analytickej metódy a následne sa aplikuje empirický prístup.
- ***ST*** – odvodené od slov „slack time“. *ST* sa dá vypočítať podľa vzťahu

$$ST = D - C. \tag{3}$$

  Z toho vyplýva, že nám stačí z týchto troch hodnôt poznať iba ľubovoľné dve a tretiu môžeme vypočítať. Každá úloha má vlastnú hodnotu *ST*.
- ***P*** – priorita úlohy. Podľa tejto hodnoty sa neurčuje poradie plnenia úloh, ale poradie výberu úlohy, ktorú vyradíme zo zoznamu naplánovaných úloh v prípade preťaženia systému. To nám umožňuje špecifikovať, ktoré úlohy sú typu hard RT, ktoré sú firm RT a ktoré nie sú vôbec RT. Čím vyššia je hodnota P, tým menej dôležité je splnenie úlohy do špecifikovaného času odozvy. Hard RT úlohy majú *P* rovné nule.
- ***ID*** – identifikátor úlohy. Každá úloha má vlastné unikátne *ID*.
- ***LD*** – odvodené od slov „latest deadline“, čiže ide o doposiaľ najväčšiu dobu odozvy zo všetkých aktuálne naplánovaných úloh. Iniciálna hodnota *LD* musí byť 0. Ak je nová úloha akceptovaná a teda naplánovaná a zároveň ak je *D* väčšie ako *LD*, tak *LD* priradíme hodnotu *D*.

- ***A*** – odvodené od slova „addition“, čiže sa jedná o prírastok novej úlohy k preťaženosti systému v prípade, že sa úloha akceptuje na naplánovanie. Túto hodnotu môžeme vypočítať ako $A = LD - ST$. Každá úloha má vlastné A. Len pre zaujímavosť:
  - $A < 0$ znamená, že naplánovanie danej úlohy znižuje preťaženosť systému.
  - $A = 0$ znamená, že naplánovanie danej úlohy neovplyvňuje preťaženosť systému.
  - $A > 0$ znamená, že naplánovanie danej úlohy prispieva k preťaženosti systému.
- ***O*** – odvodené od slova „overload reserve“, čiže sa jedná o rezervu k preťaženosti systému. Preťaženosť vypočítame ako sumu hodnôt A všetkých naplánovaných úloh. Rezerva k preťaženosti je opačná hodnota ako preťaženosť. Ideálne je pamätať si hodnotu O a postupne od tejto hodnoty odpočítavať hodnotu A novej úlohy. Ak je O záporné, tak systém je preťažený a musí sa dostať z tohto stavu výberom úlohy, ktorú odstráni zo zoznamu naplánovaných úloh a zaradí do zoznamu odmietnutých úloh.

Teraz si ukážeme správanie algoritmu FED po prijatí požiadavky na naplánovanie novej úlohy (obr. 3.1).



Obr. 3.1: Spracovanie novej úlohy algoritmom FED (diagram aktivít)

Po získaní požiadavky na naplánovanie novej úlohy najprv vypočítame *A* a *O*, a pridáme novú úlohu do zoznamu naplánovaných úloh. Následne sa pozrieme na hodnotu *O*, podľa ktorej zistíme, či je systém preťažený. Ak áno, tak nájdeme obeť, čiže úlohu, ktorú odplánujeme. Výber obete je najprv podľa hodnoty *P* a následne ešte podľa *A*. Výber obetí opakujeme, až kým neprestane byť systém preťažený. Nakoniec iba aktualizujeme *LD*.

Po skončení práve vykonávanej úlohy odstránime skončenú úlohu zo zoznamu naplánovaných úloh. Okrem toho tiež aktualizujeme hodnoty *D* a *A* všetkých naplánovaných aj nenaplánovaných úloh spolu s hodnotou *LD* a *O*. Po aktualizácii týchto hodnôt ešte vyberieme takú úlohu zo zoznamu nenaplánovaných úloh, ktorá má najmenšie *P* a následne najmenšie *A*. Vybranú úlohu znovu naplánujeme, ak platí vzťah:

$$O + A \geq 0 \quad (4)$$

Vyššie uvedená podmienka nám zabezpečuje, že opätovné naplánovanie vybranej úlohy nespôsobí preťaženie systému, pretože po odčítaní hodnoty *A* od *O* zostane hodnota *O* väčšia alebo rovná 0. Pokiaľ platí táto podmienka (t.j. systém nie je preťažený) a zároveň platí, že úlohy sú zoraďované podľa ich doby odozvy (rovnako ako v prípade algoritmu EDF), tak máme zaručené splnenie stanovených dôb odozvy naplánovaných úloh.

Keďže prvé kritérium, podľa ktorého vyberáme úlohy zo zoznamu naplánovaných úloh (a aj spätne vkladáme), je priorita úloh a zároveň hard RT úlohy majú prioritu $P = 0$, tak vieme zaručiť, že hard RT úlohy sú vždy vyradené zo zoznamu naplánovaných úloh až ako posledné. Z toho vyplýva, že všetky hard RT úlohy majú zaručené splnenie doby svojej odozvy, pokiaľ je to možné (možné to nie je iba v tom prípade, ak neexistuje vôbec žiadne poradie úloh, ktoré by teoreticky bolo schopné splniť všetky úlohy včas).

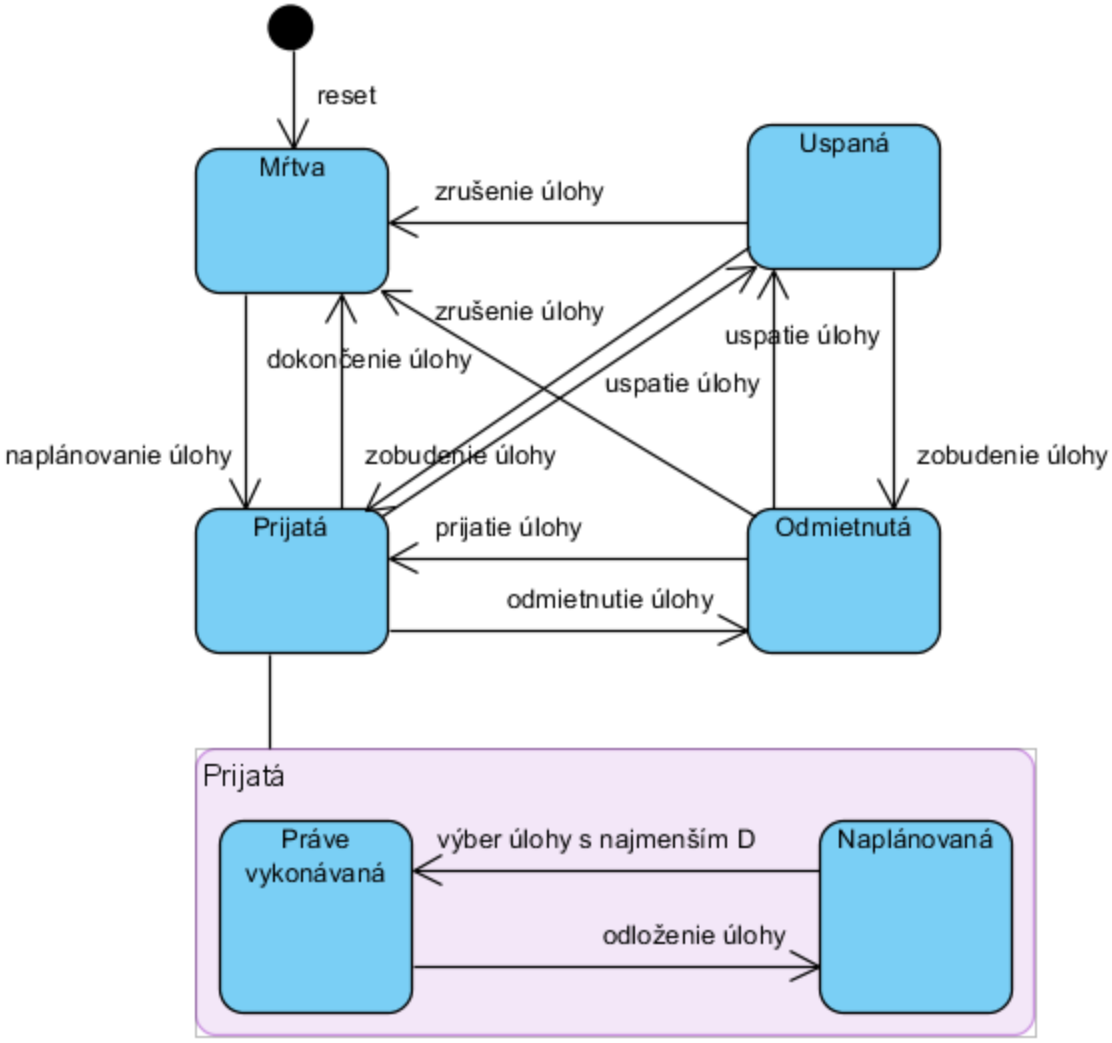
3.2.2 Stavy úloh pre algoritmus FED

Každá úloha sa nachádza práve v jednom z týchto stavov:

- *Mŕtva* – tento stav reprezentuje prázdnu bunku v zozname úloh, ide teda o neexistujúcu úlohu (t.j. nebola ešte naplánovaná funkciou *naplánovanie úlohy* alebo úloha už bola splnená, prípadne zrušená).
- *Prijatá* – úloha, ktorá prešla testom preťaženia systému a teda bude splnená. Podľa hodnoty *D* delíme prijaté úlohy na:
    - *Aktívna* – práve vykonávaná úloha, t.j. taká prijatá úloha, ktorá má najmenšie *D* a preto ju vykonáva procesor.
    - *Naplánovaná* – úloha, ktorá sa nachádza v zozname naplánovaných úloh a pokiaľ zostane v tomto zozname dostatočne dlho, tak sa stane práve vykonávanou úlohou.
- *Odmietnutá* – úloha bola vyradená zo zoznamu naplánovaných úloh v dôsledku preťaženia systému.
- *Uspaná* – pozastavená úloha, ktorá bola vyradená zo zoznamu naplánovaných úloh v dôsledku zavolania funkcie *Uspatie úlohy*.

Na obrázku 3.2 môžeme vidieť stavový diagram, ktorý opisuje možné prechody z jedneho stavu úlohy do ostatných stavov. Ako vidno z tohto diagramu, počiatočný stav je *mŕtvy*.

Funkciou *naplánovanie úlohy* sa pokúšame o to, aby plánovač úloh danú úlohu naplánoval a teda aby sa úloha vykonala a splnila v stanovenej dobe odozvy. Preto sa úloha vloží do zoznamu prijatých úloh a teda zmeníme stav úlohy na *prijatú*. Ak sa stane systém preťažený, tak úlohu odložíme do zoznamu odmietnutých úloh a danej úlohe zmeníme stav na *odmietnutú*. Po predčasnom dokončení práve vykonávanej úlohy môžeme opätovne prijať odmietnutú úlohu, vďaka čomu táto úloha získa znova tav *prijatá*. Ak má táto úloha najmenšie *D*, tak sa môže stať priamo *práve vykonávaná*, inak je medzitým v stave *naplánovaná*. Úlohy v stave *prijaté* alebo *odmietnuté* by sme mohli ešte generalizovať ako úlohy, ktoré nie sú v stave *uspané*. Ak úloha nie je v stave *uspaná* (t.j. ak je úloha *prijatá* alebo *odmietnutá*), tak môže byť *uspaná* funkciou *uspanie úlohy*. Následne môže byť *uspaná* úloha zobudená funkciou *zobudenie úlohy*. Úloha sa môže dostať do stavu *mŕtva* buď tým, že sa vykonávanie úlohy dokončí a teda úloha je splnená, alebo tým, že sa nesplní stanovená doba odozvy úlohy klesnutím hodnoty *ST* pod 0, ak ide o úlohu. Výnimku tvoria non-RT úlohy, ktoré po dosiahnutí hodnoty ST rovnej 0 nie sú zrušené, ale iba im resetujeme hodnotu *D* na pôvodnú iniciálnu hodnotu. Z toho vyplýva, že non-RT úlohy môžu „umrieť“ (t.j. dostať sa do stavu *mŕtve*) iba použitím funkcie *odstránenie úlohy*.



Obr. 3.2: Možné prechody stavov úloh hardvérového plánovača (stavový diagram)

3.2.3 Architektúra hardvérového plánovača FED

Hardvérový plánovač realizujúci algoritmus FED by mal obsahovať minimálne tieto časti:

- **Riadiaca jednotka:**
  - zabezpečuje komunikáciu hardvérového plánovača s rozhraním FSL

- dekóduje inštrukcie, ktoré predstavujú funkcie vymenované v sekcii 3.1.1 Funkcionálne požiadavky (s. 24)
- riadi správanie ostatných častí hardvérového plánovača

- **Banka úloh:**
  - zoznam všetkých úloh indexovaný podľa hodnoty *ID*
  - každá bunka tejto štruktúry reprezentujúca jednu úlohu obsahuje stav úlohy a iniciálne hodnoty potrebné pre naplánovanie úlohy (t.j. *P*, *D* a *C*)
- **Zoznam naplánovaných úloh:**
  - tento komponent poskytuje 2 funkcie:
    - vložiť novú úlohu do zoznamu a zaradiť ju na takú pozíciu, aby boli úlohy zoradené podľa hodnoty *D*, pričom výstup tohto obvodu vždy vypisuje *ID* úlohy s najmenšou hodnotou *D*
    - odstrániť ľubovoľnú úlohu zo zoznamu na základe prijatej hodnoty *ID*
  - navyše by mal tento komponent automaticky aktualizovať hodnoty *D*, *ST*, *A* a poradie úloh s tým, že ak klesne hodnota *ST* na zápornú hodnotu, tak sa daná úloha odstráni zo zoznamu.
- **Jednotka preťaženia:**
  - uchováva a aktualizuje hodnotu *O*, a kontroluje stav preťaženia systému
  - obsahuje zoznam prijatých úloh zoradených podľa zreťazenia hodnôt *T* a *A*. Napríklad pre hodnoty *T* = 100 a *A* = 11010 dostaneme zreťazenú hodnotu, ktorá je 10011010.
  - obsahuje zoznam odmietnutých úloh, t.j. úlohy ktoré boli odložené v dôsledku preťaženia systému. Tento zoznam je taktiež zoradený podľa zreťazenia hodnôt *T* a *A*, ale s tým rozdielom, že výstupom zoznamu nie je *ID* úlohy s najväčšou zreťazenou hodnotou *T* a *A*, ale najmenšou.

3.2.4 Architektúra hardvérového plánovača EDF

Rozhodli sme sa nakoniec nepoužiť nás nový algoritmus FED z týchto dôvodov:

- Algoritmus FED síce umožňuje kombinovať hard RT úlohy aj s firm RT a non RT úlohami, ale za cenu minimálne 4-násobne väčšej hardvérovej architektúry a teda aj spotreby plochy na čipe v porovnaní s plánovačom realizujúcim EDF.
- Zatiaľ čo algoritmus EDF je možné realizovať konštantným počtom taktov, algoritmus FED by požadoval variabilný počet taktov kvôli riešeniu preťaženia systému. Keď sa systém preťaží, FED postupne odkladá niekoľko úloh. Dopredu však nevieme koľko a práve toto oslabuje deterministické správanie systému.

- Algoritmus EDF je aj tak najvhodnejší pre hard RT systémy, ktoré obsahujú výhradne hard RT úlohy. A práve pre takéto systémy je hardvérová akcelerácia najväčším prínosom.

Preto sme navrhli hardvérový plánovač úloh, ktorý realizuje algoritmus EDF. Tento plánovač sme však ešte rozšírili o schopnosť odstrániť ľubovoľnú úlohu na základe jej *ID*, nakoľko existujúce riešenia umožňujú odstrániť vždy iba úlohu s najmenšou dobou odozvy (t.j. práve vykonávanú úlohu).

Hardvérový plánovač úloh je navrhnutý v podobe koprocesora, ako už bolo špecifikované v kapitole 3.1 Špecifikácia požiadaviek. Koprocesory používajú štandardne dva registre na vstupné dáta a následne zapíšu svoj výsledok do tretieho registra. Rovnaké rozhranie používa aj náš koprocesor. Taktiež sme si overili, že presne takéto rozhranie sa používa napríklad pri CPU Altera Nios II.

Vstupy do tohto koprocesora sú:

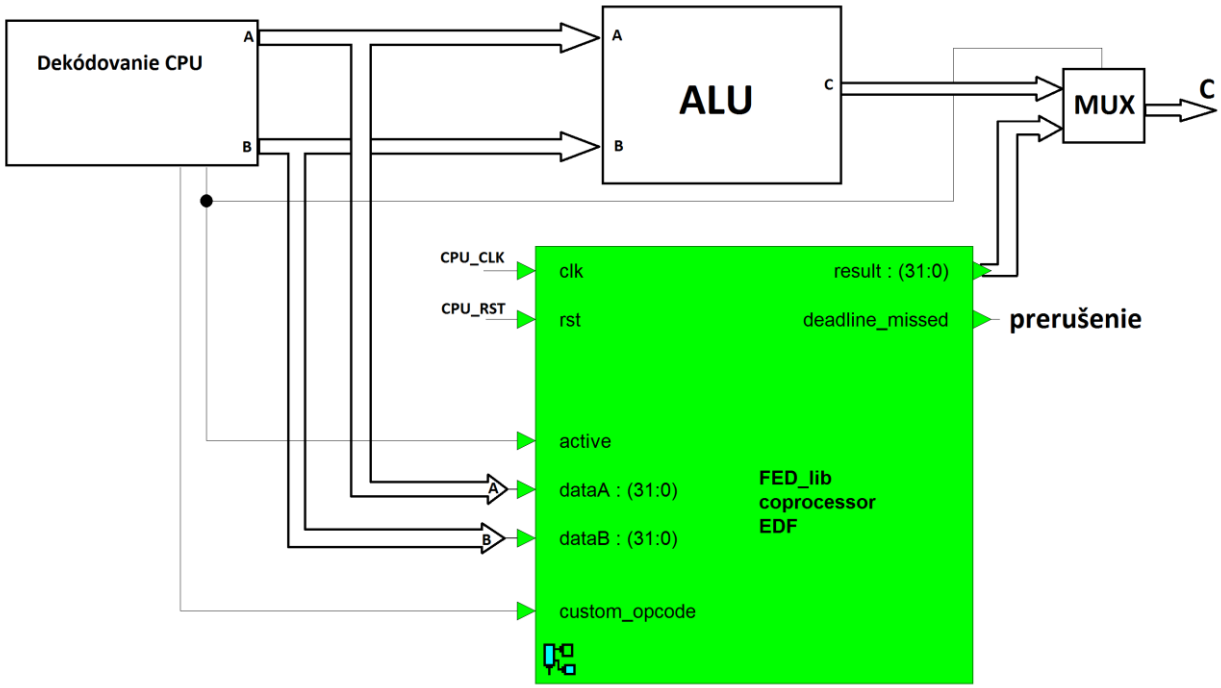
- *CLK* – synchronizačný hodinový signál pre preklápacie obvody
- *RST* – resetujúci signál na inicializáciu koprocesora
- *active* – signál určujúci, či je práve vykonávaná inštrukcia určená pre tento koprocesor
- *custom_opcode* – signál, ktorý určuje, ktorú inštrukciu má koprocesor vykonať
- *dataA* – dáta z registra A obsahujúce *ID* úlohy
- *dataB* – dáta z registra B obsahujúce *deadline* úlohy

Výstupy z tohto koprocesora sú:

- *result* – dáta do registra C obsahujúce *ID* aj *deadline* tej úlohy, ktorá má najmenší *deadline* (t.j. úlohy, ktorá sa má práve vykonávať)
- *deadline_missed* – signál prerušenia, ktorý informuje CPU o tom, či je prekročená doba odozvy úlohy s najmenšou dobou odozvy

Rozhranie koprocesora môžeme vidieť na obrázku 3.3, ktorý je na nasledujúcej strane. Ako vidno z tohto obrázka, koprocesor je riadený dekódovaním CPU signlom *active* a *custom_opcode*, ktoré dekódovacia jednotka CPU generuje na základe dekódovanej inštrukcie. Koprocesor rozširuje aritmeticko-logickú jednotku a preto sú výstupy oboch

jednotiek multiplexované na základe signálu *active*. Údaje *dataA* a *dataB* získava koprocesor rovnakým spôsobom ako ALU procesora. Taktiež aj signály *clk* a *rst* sú rovnaké ako pre CPU.



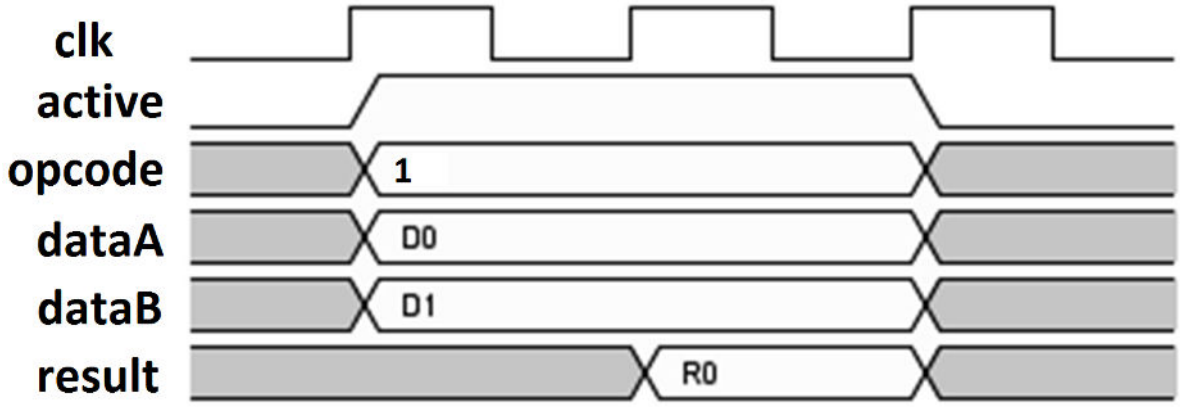
Obr. 3.3: Rozhranie koprocesora

Náš koprocesor obsahuje 2 inštrukcie a 1 pseudoinštrukciu:

- *task_schedule* – inštrukcia, ktorá vloží novú úlohu do koprocesora a naplánuje túto úlohu na základe jej maximálnej doby odozvy (*deadline*).
- *task_kill* – inštrukcia, ktorá odstráni ľubovoľnú úlohu z koprocesora na základe *ID* úlohy.
- *NOP* – pseudoinštrukcia, ktorá nevykoná žiadnu operáciu. Realizujeme ju pomocou inštrukcie *task_schedule*, pričom *deadline* je rovný maximálnej možnej hodnote (t.j. binárne samé jednotky).

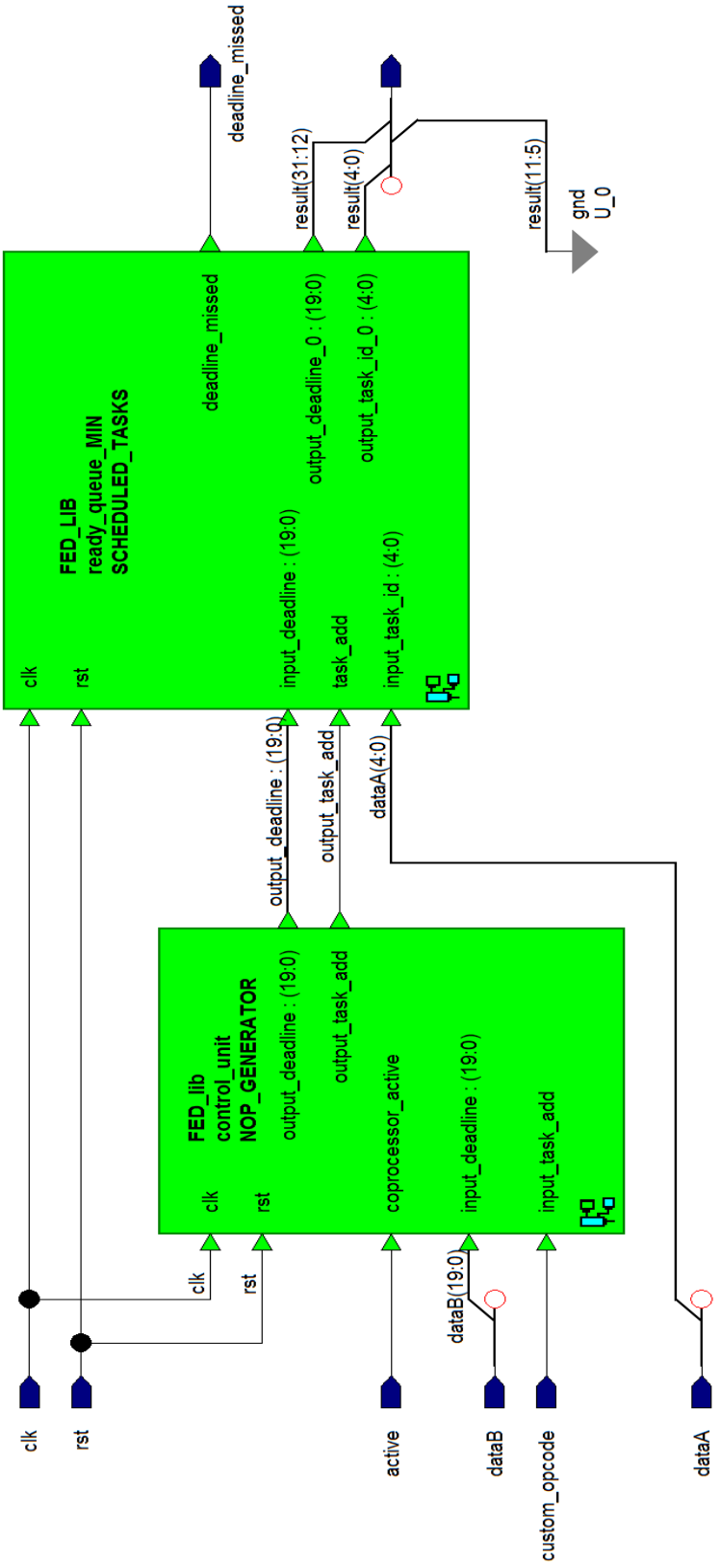
Každá inštrukcia vypisuje na výstup *result* úlohu, ktorú má CPU práve vykonávať. V prípade, že chceme zistiť, ktorá úloha sa má vykonávať a pritom nechceme meniť obsah plánovača úloh, môžeme na to použiť práve inštrukciu *NOP*. Z hľadiska časovania je realizovaná každá inštrukcia na 2 takty bez ohľadu na aktuálny počet úloh v koprocesore

a kapacitu (veľkosť) koprocesora. Počas prvého taktu sa aplikuje zmena v zozname naplánovaných úloh na základe použitej inštrukcie. Druhý takt je použitý na generovanie výstupu koprocesora na základe nového poradia úloh. Časový priebeh inštrukcií koprocesora je znázornený na nasledovnom obrázku (obr. 3.4).



Obr. 3.4: Časový priebeh inštrukcií koprocesora

Na najvyššej úrovni abstrakcie môžeme koprocesor rozdeliť na dve základné časti: riadiaca jednotka a operačná časť. Aj v prípade nášho koprocesora je systém rozdelený na tieto dve časti. Riadiaca jednotka prijíma informácie z vonkajšieho prostredia a na základe týchto informácií riadi operačnú časť. Tieto informácie sú konkrétne signál *active*. V prípade, že *active* = ‘0‘, tak riadiaca jednotka generuje pre operačnú časť pseudoinštrukciu *NOP*. Túto pseudoinštrukciu musí riadiaca jednotka generovať aj pre druhý takt ľubovoľnej inštrukcie koprocesora. Operačná časť je v našom prípade zoznam naplánovaných úloh, ktorý je podrobnejšie opísaný v nasledujúcej podkapitole. Štruktúru koprocesora na najvyššej úrovni abstrakcie môžeme vidieť na obrázku na ďalšej strane (obr. 3.5).

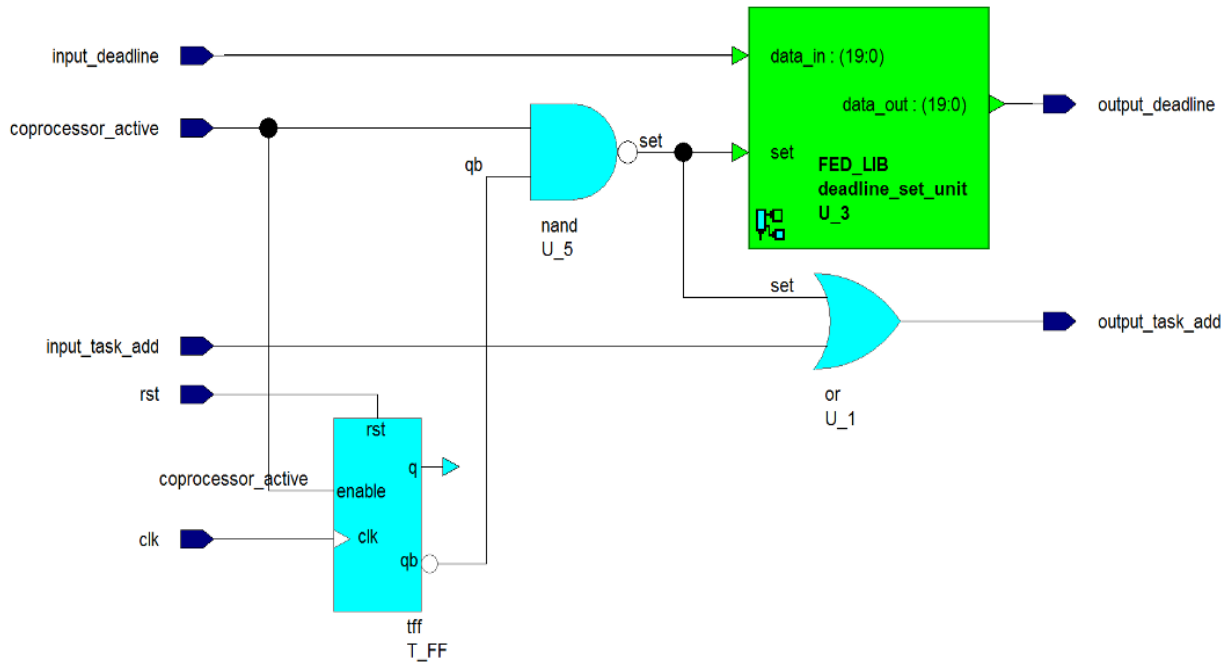


Obr. 3.5: Štruktúra koprocesora

3.2.5 Architektúra riadiacej jednotky

Ako už bolo spomenuté na predchádzajúcej strane, riadiaca jednotka je zodpovedná za riadenie zoradeného zoznamu úloh. Takéto riadenie má iba jedinú úlohu. Generovať pseudoinštrukciu *NOP* vtedy, keď sa nemá meniť zoznam úloh.

Štruktúra riadiacej jednotky je znázornená na obr. 3.6, ktorý je nižšie. Obsahuje jeden T preklápací obvod, ktorý reprezentuje 1-bitové počítadlo. Toto počítadlo sa používa na rozlíšenie toho, či sa jedná o prvý alebo druhý takt inštrukcie. Ďalej sa nachádza v riadiacej jednotke jedno NAND hradlo, ktoré generuje signál *set*. Tento signál vytvára pseudoinštrukciu *NOP* tým, že nastavuje *output_deadline* a *output_task_add* na samé jednotky pomocou viacerých OR hradiel zapojených paralelne. Celkový počet OR hradiel v riadiacej jednotke je rovný šírke vektora *output_deadline* + 1.



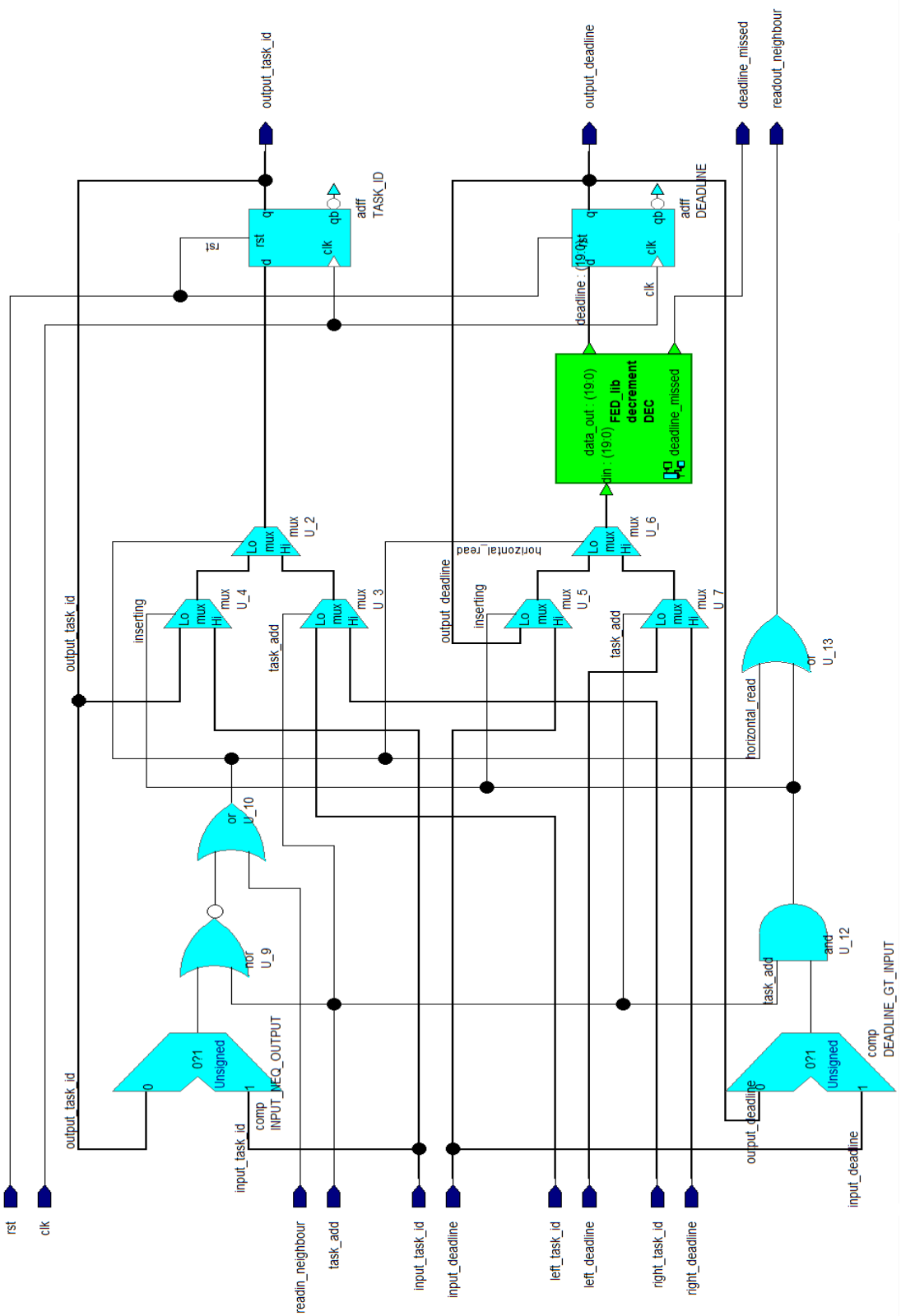
Obr. 3.6: Štruktúra riadiacej jednotky

3.2.6 Architektúra zoznamu naplánovaných úloh

Plánovač úloh potrebuje zoraďovať úlohy podľa jediného kritéria, ktorým je *deadline* úlohy. Tento *deadline* je v našej architektúre vnímaný ako časový interval medzi súčasnosťou a termínom, dokedy musí byť úloha dokončená. Jedná sa teda o veličinu, ktorá sa postupne dekrementuje smerom k nule. Keď klesne až na nulu, tak uplynul všetok čas vyhradený pre danú úlohu. V rámci tejto práce sme sa zaoberali aj tým, koľko bitov je vhodné použiť na reprezentáciu časového intervalu *deadline*. V prípade, že pre maximálnu hodnotu *deadline* nám postačuje 1 sekunda a zároveň potrebujeme odchýlku presnosti času 1 mikrosekunda, tak je ideálne použiť 20 bitov, pretože 1 sekunda obsahuje milión mikrosekúnd a na hodnotu 1 milión potrebujeme aspoň 20 bitov.

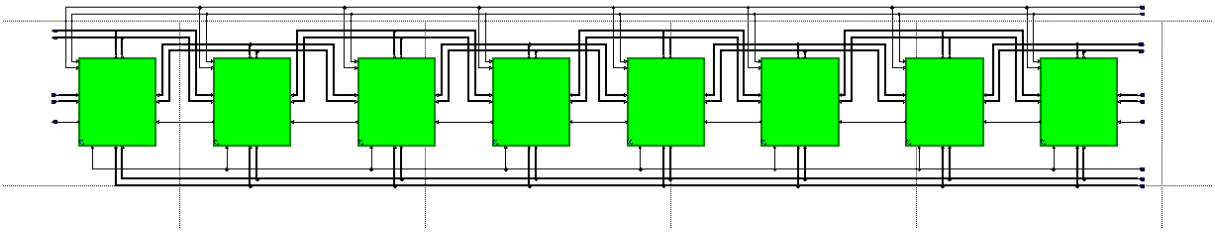
Architektúra zoraďovania úloh hrá významnú úlohu v celej architektúre plánovača úloh, pretože predstavuje väčšinu systému z hľadiska spotrebovanej logiky na čipe. Za účelom dosiahnuť čo najefektívnejšiu štruktúru plánovača úloh sme pre realizáciu zoraďovania úloh použili posuvné registre so spoločným paralelným vstupom vylepšené o techniku prúdového spracovania inštrukcií (známe ako *pipelining*). Návrh sme robili postupne smerom zdola nahor. Na najnižšej úrovni abstrakcie sa nachádza opis jednej bunky, ktorá predstavuje kapacitu jednej úlohy.

Na obrázku 3.7 môžeme vidieť štruktúru jednej bunky posuvných registrov. Ako vidno z tohto obrázka, bunka obsahuje dva registre, horný uchováva *ID* úlohy a dolný uchováva *deadline* úlohy. Na obrázku sa ďalej nachádzajú multiplexorové stromy, ktoré určujú to, ktorá úloha sa uloží do danej bunky. Existujú 4 možnosti: úloha zo spoločného vstupu, úloha nachádzajúca sa už v danej bunke, úloha v bunke naľavo alebo úloha v bunke napravo. Okrem toho sa nachádza v každej bunke jeden komparátor, ktorý porovnáva *ID* úlohy, jeden komparátor, ktorý porovnáva *deadline* úlohy a jeden dekrementujúci obvody, ktorý automaticky dekrementuje *deadline* úlohy. Vstupný signál *task_add* určuje, či sa úloha na vstupe má pridať alebo odobrať zo zoznamu úloh.



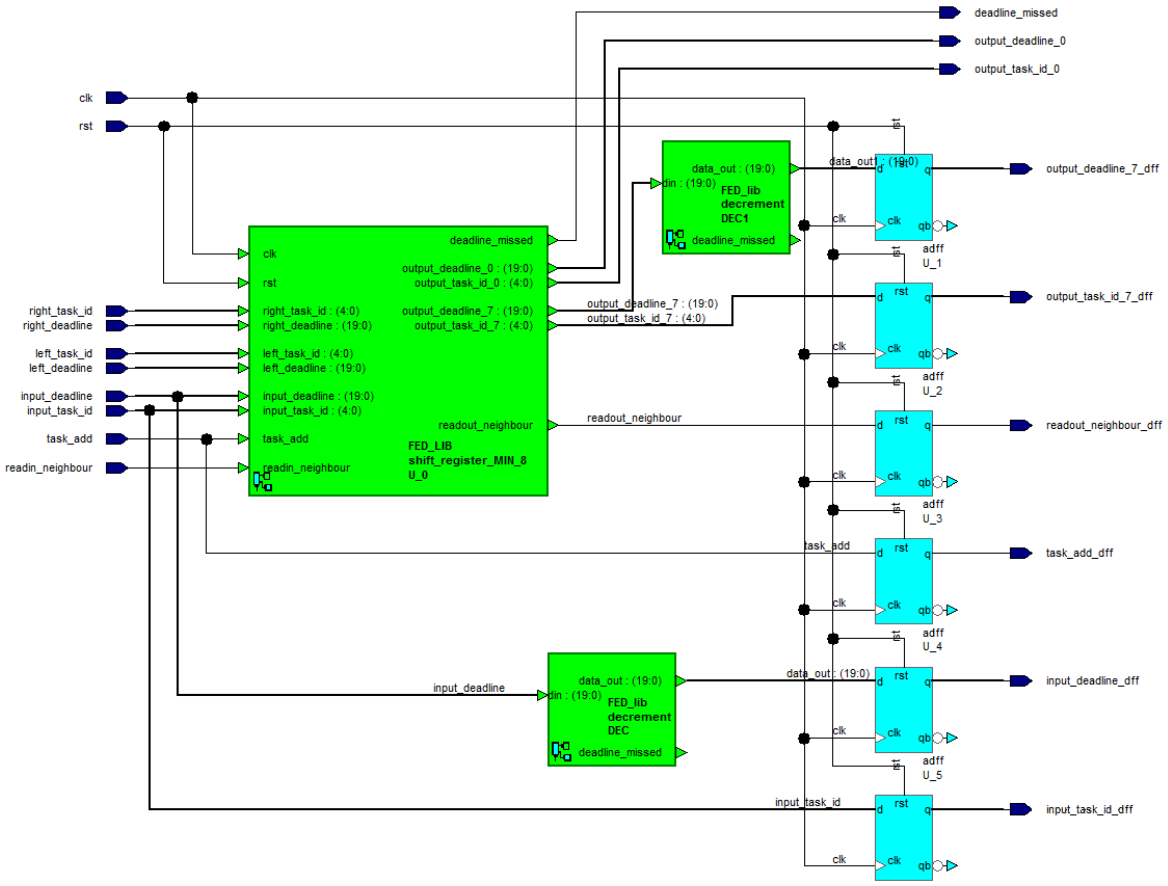
Obr. 3.7: Štruktúra jednej bunky predstavujúcej jednu úlohu

Na obrázku 3.8 môžeme vidieť zapojenie ôsmich buniek do posuvných registrov so spoločným paralelným vstupom. Ako vidno z tohto obrázka, spoločný vstup je vedený v dolnej časti obrázka. Okrem toho bunky komunikujú navzájom prostredníctvom susedných buniek v rámci jedného hodinového cyklu. Takéto zapojenie je výhodné z hľadiska spotreby plochy na čipe, ale treba si uvedomiť, že čím viac buniek by sme takto zapojili, tým dlhšia by bola kritická cesta kvôli spoločnej zbernici na vstupe. Kritická cesta nám ovplyvňuje maximálnu možnú frekvenciu, s akou môže koprocesor pracovať. Keďže kritická cesta koprocesora by mala byť kratšia alebo nanajvýš rovnako dlhá ako kritická cesta CPU, tak sme sa rozhodli paralelne zapojiť iba 8 buniek.



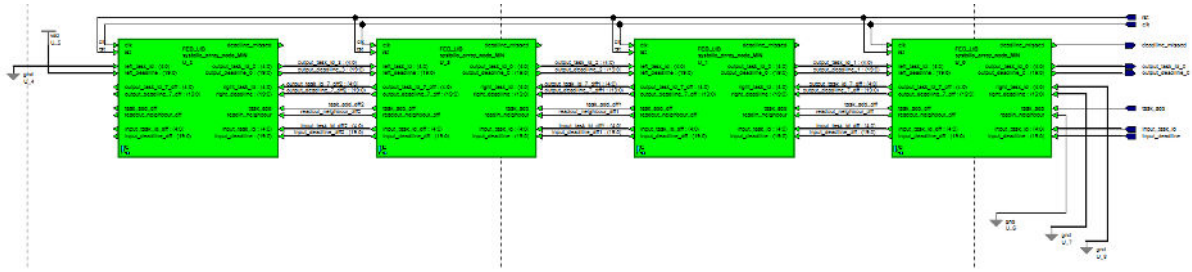
Obr. 3.8: Štruktúra posuvného registra s paralelným vstupom

Na realizáciu väčšieho počtu buniek používame *pipelining*. Každých osem buniek tvorí jednu vrstvu *pipelining*-u, známu ako pipeline stage. Na obrázku 3.9 vidíme štruktúru jedného pipeline stage bloku. Táto štruktúra obsahuje 8 buniek zapojených paralelne a medzivrstvu, ktorú je nutné pridať medzi každé dve takéto vrstvy. Takáto medzivrstva je zložená zo skupiny preklápacích obvodov a dvoch dekrementačných jednotiek potrebných pre aktualizáciu *deadline* hodnôt. Takáto architektúra má výhodu v tom, že maximálny počet úloh v systéme je možné jednoducho zvýšiť pridaním ďalších pipeline stage blokov, pričom nám vôbec nenarastá oneskorenie v obvode.



Obr.: 3.9: Štruktúra bunky sériovej architektúry

Na obrázku 3.10 vidíme štruktúru zoznamu naplánovaných úloh, ktorá obsahuje 4 pipeline stage bloky. Tento zoznam úloh je orientovaný tak, že napravo je vstup aj výstup zoznamu, takže aj úloha s najmenším *deadline* sa nachádza najviac napravo. A práve táto úloha je úloha, ktorá sa má vykonávať. Keďže každý blok pipeline stage má kapacitu 8 úloh a tieto bloky sú v tejto štruktúre 4, tak celková kapacita zoznamu naplánovaných úloh je 32 úloh.



Obr.: 3.10: Štruktúra zoznamu naplánovaných úloh

3.2.7 Teoretická spotreba plochy na čipe

Na základe navrhnutého koprocesora je možné teoreticky vyjadriť množstvo logiky potrebnej pre realizáciu takéhoto koprocesora v nejakej technológii ASIC obvodov. Skôr, než si vyjadríme toto množstvo, mali by sme si uvedomiť, od čoho závisí:

- *N* – počet buniek v zozname úloh, t.j. maximálny počet úloh v systéme
- *PSC* – počet „pipeline stage“ buniek
- *IDw* – šírka (počet bitov) čísla *ID*
- *Dw* – šírka (počet bitov) čísla *deadline*

Riadiaca jednotka predstavuje minimálne, dokonca až zanedbateľne malé množstvo logiky v porovnaní so zoznamom úloh. Napriek tomu sme určili aj náklady na realizáciu riadiacej jednotky. Tabuľka 1 obsahuje teoretickú spotrebu plochy pre jednotlivé časti systému. *CU* („Control Unit“) predstavuje logiku potrebnú pre kontrolnú jednotku. *TC* („Task Cell“) predstavuje logiku potrebnú pre jednu bunku/úlohu. *PSI* („Pipeline Stage Interconnect“) predstavuje réžiu prepojenia dvoch „pipeline stage“ blokov. *TAC* („Total Area Cost“) vyjadruje celkovú spotrebu plochy pre celý systém v prípade, keď $N = 32$, $PSC = 4$, $IDw = 5$ a $Dw = 20$.

	CU	*TC*	*PSI*	*TAC*
D-FF	1	$IDw + Dw$	$2\,IDw + 2\,Dw + 2$	957
MUX	1	$3\,IDw + 3\,Dw$	0	2401
NAND	1	0	0	1
NOR	0	1	0	32
AND	0	1	0	32
OR	$Dw + 1$	2	0	85
XOR	0	IDw	0	160
COMP1	0	1	0	32
COMP2	0	1	0	32
DEC	0	1	2	38

Tabuľka 1: Teoretická spotreba plochy pre jednotlivé časti systému

Celková spotreba plochy pre celý systém je vyjadrená nasledovným vzorcom:

$$TAC = CU + N.TC + (PSC - 1)\,.\,PSI \qquad (4)$$

Dosadením hodnôt z tabuľky 1 do vzorca (4) dostávame detailnejšie vyjadrenie celkovej plochy v tabuľke 2.

	TAC
D-FF	(*N* + 2 . *PSC* - 2) . (*IDw* + *Dw*) + 2 . *PSC* - 1
MUX	3 . *N* . (*IDw* + *Dw*) + 1
NAND	1
NOR	*N*
AND	*N*
OR	2 . *N* + *Dw* + 1
XOR	*N* . *IDw*
COMP1	*N*
COMP2	*N*
DEC	*N* + 2 . *PSC* - 2

Tabuľka 2: Celková spotreba plochy pre celý systém

Ďalej by bolo možné ísť až na úroveň tranzistorov, keďže počet tranzistorov pre jednotlivé hradlá je všeobecne známy. Problém však je, že tento počet tranzistorov sa už môže mierne líšiť v závislosti od použitej technológie (napr. CMOS, NMOS, Bi-CMOS, ...). Odlišná spotreba plochy na čipe môže nastať aj kvôli tomu, že komparátory COMP1 a COMP2, ako aj dekrementačný obvod DEC boli opísané správaním, nie štruktúrou. Preto sme nepristúpili k vyjadreniu spotreby plochy až na úrovni počtu tranzistorov. Veríme však, že vďaka vyššie uvedeným vzorcom v tabuľke 2 bude jednoduchšie dodatočné vyjadrenie počtu tranzistorov na základe výberu konkrétnej technológie a výberu konkrétnej štruktúry COMP1, COMP2 a DEC.

3.3 Overenie riešenia

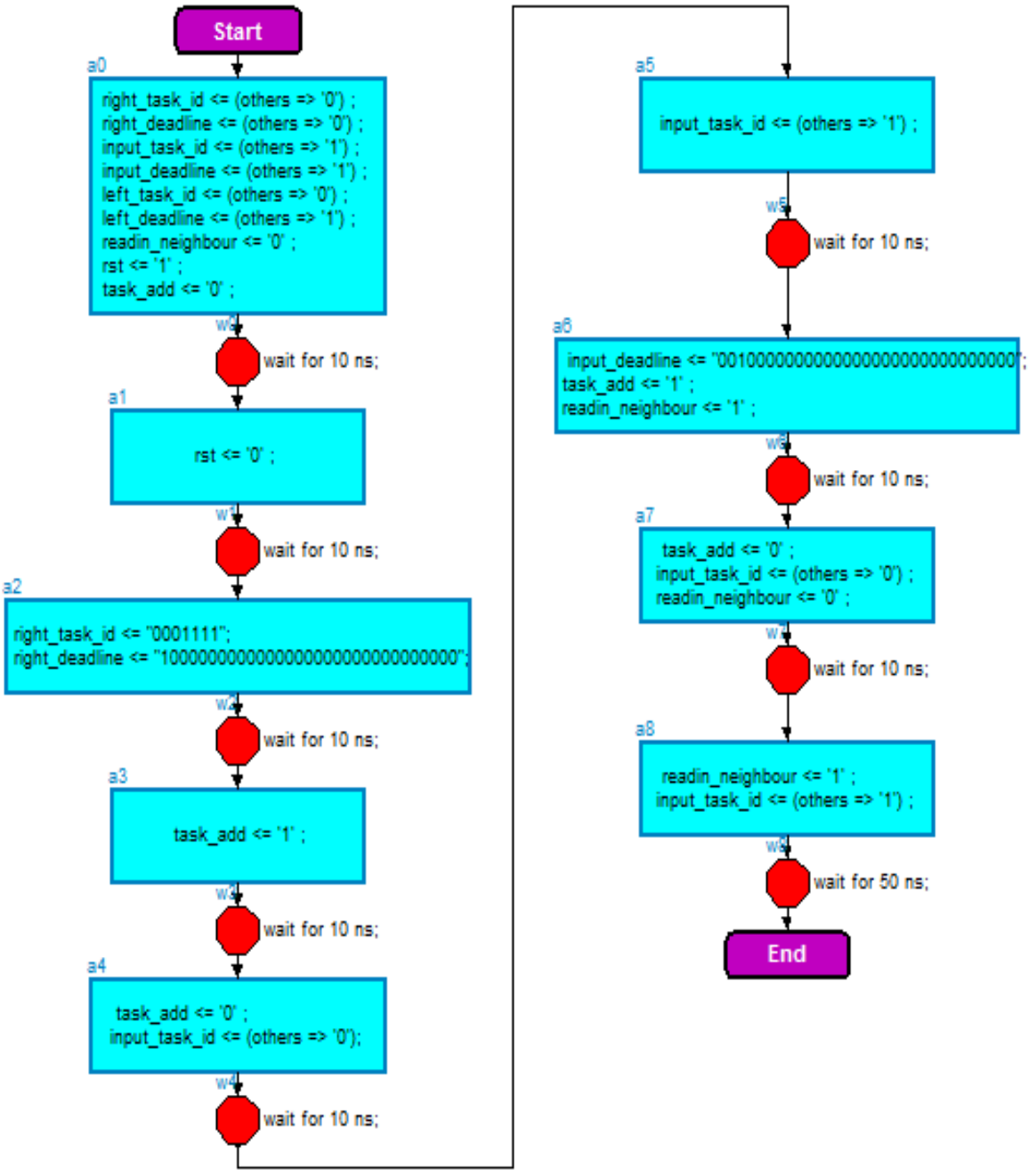
Navrhnuté riešenie bolo overované postupne počas jednotlivých etáp návrhu a boli požité viaceré spôsoby overenia riešenia. Najprv sme overili jednu bunku reprezentujúcu jednu úlohu formou simulácie v ModelSim-e, pričom sme kombinovali rôzne postupnosti vkladania a mazania úloh. Následne sme overili rovnakou simuláciou celý zoznam naplánovaných úloh. Po navrhnutí celého koprocesora sme vykonali verifikáciu koprocesora, tiež simuláciou v ModelSim-e. Nasledoval opis plánovania úloh softvérovo (v jazyku C) a porovnanie výsledkov zo softvérovej verzie plánovania s výsledkami simulácie hardvérového koprocesora. Poslednú etapu verifikácie prestavovala syntéza a implementácia koprocesora na platforme FPGA. Na tento účel bol navrhnutý funkcionálny BIST, ktorý generuje inštrukcie koprocesora a overuje korektnosť výstupu koprocesora. Výsledky testovania zobrazoval BIST na štyroch LED diódach, ktoré sa nachádzajú na vybranej platforme FPGA.

Obrázky simulácií sú z dôvodu čitateľnosti otočené o 90 stupňov v protismere hodinových ručičiek.

3.3.1 Simulácia bunky a zoznamu naplánovaných úloh

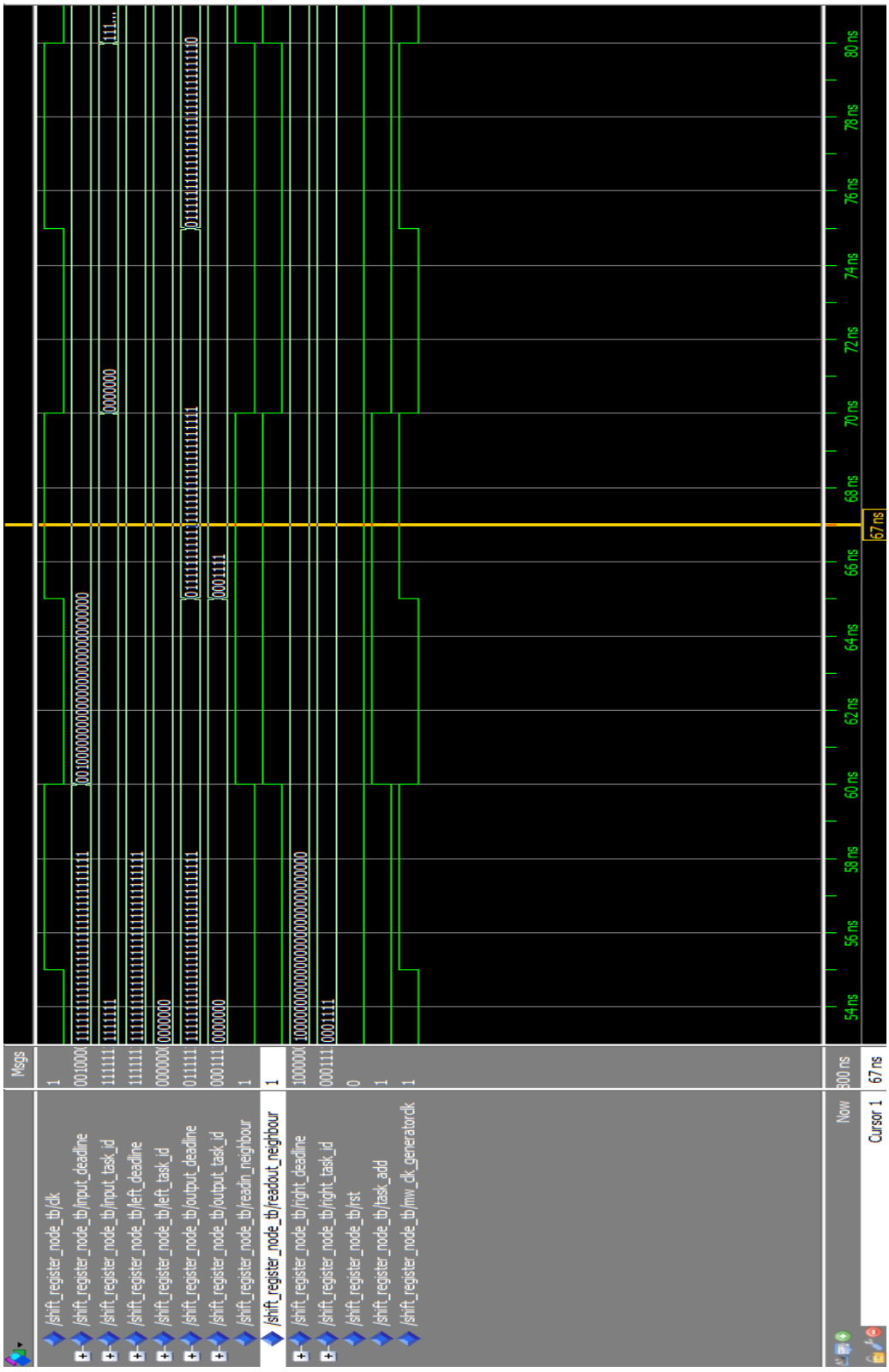
Na tieto simulácie sme použili periódu hodín s dĺžkou 10 ns, ktorá reprezentuje frekvenciu systému 100 MHz.

Ako prvé sme verifikovali simuláciou správnu funkčnosť jednej bunky posuvného registra. Testbench generuje vstupné signály podľa nasledovného obrázka (obr. 3.11). Prvých 10 ns je obvod resetovaný signálom *rst*. Ako vidíme na obrázku, overené je aj pridávanie novej úlohy aj mazanie existujúcej úlohy (ktoré riadi signál *task_add*). Taktiež je overená správna funkčnosť prenosu údajov medzi susednými bunkami.



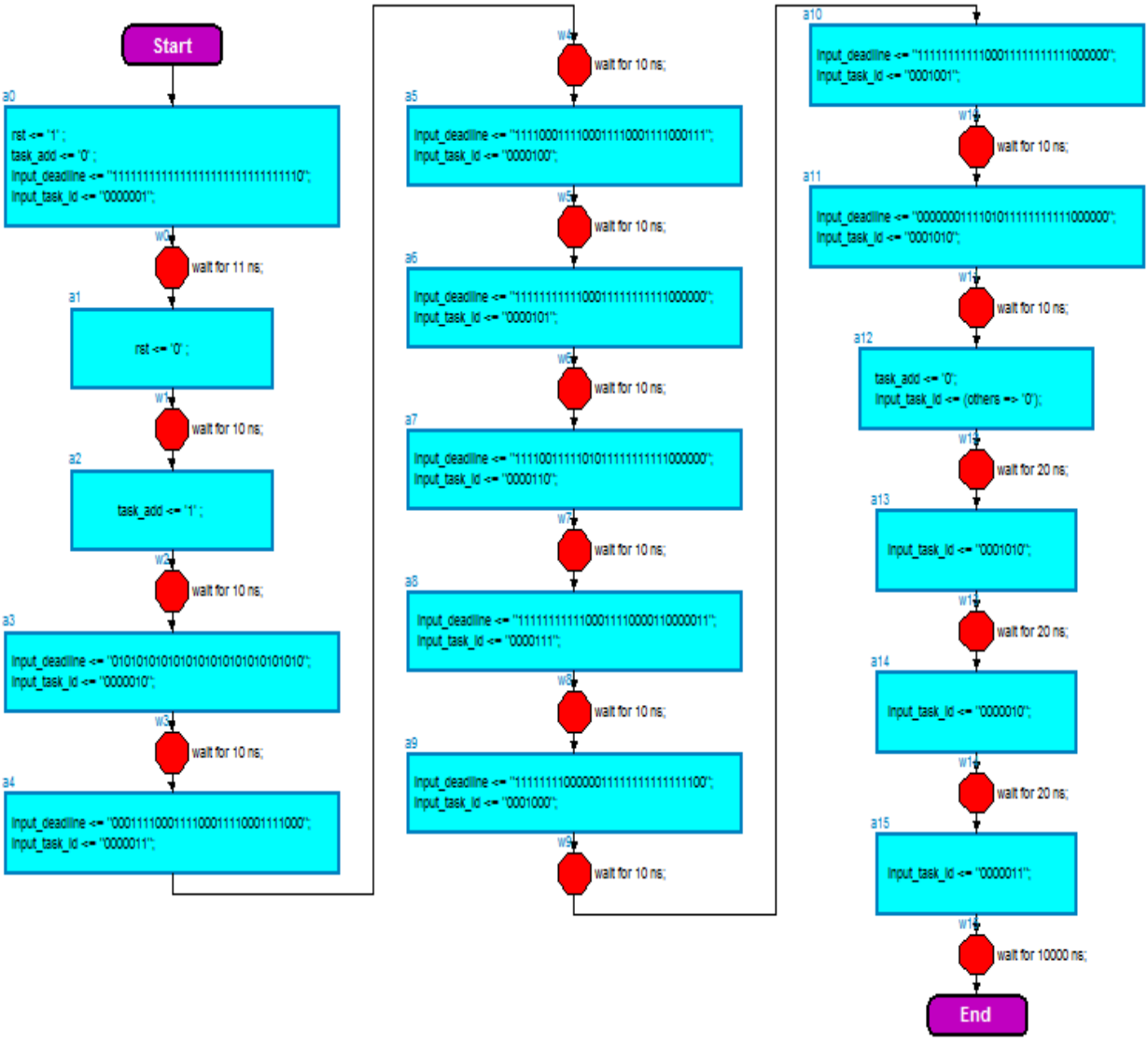
Obr. 3.11: Testovacie vstupné signály pre bunku posuvných registrov

Výsledky simulácie môžeme vidieť na obrázku 3.12. Na tomto obrázku vidíme zmenu výstupu bunky, čo predstavuje zápis úlohy do tejto bunky z pravého suseda. Keďže doba odozvy sa postupne skracuje, tak výstupná doba odozvy má o 1 menšiu hodnotu ako bola vstupná doba odozvy z pravého suseda v predchádzajúcom takte.



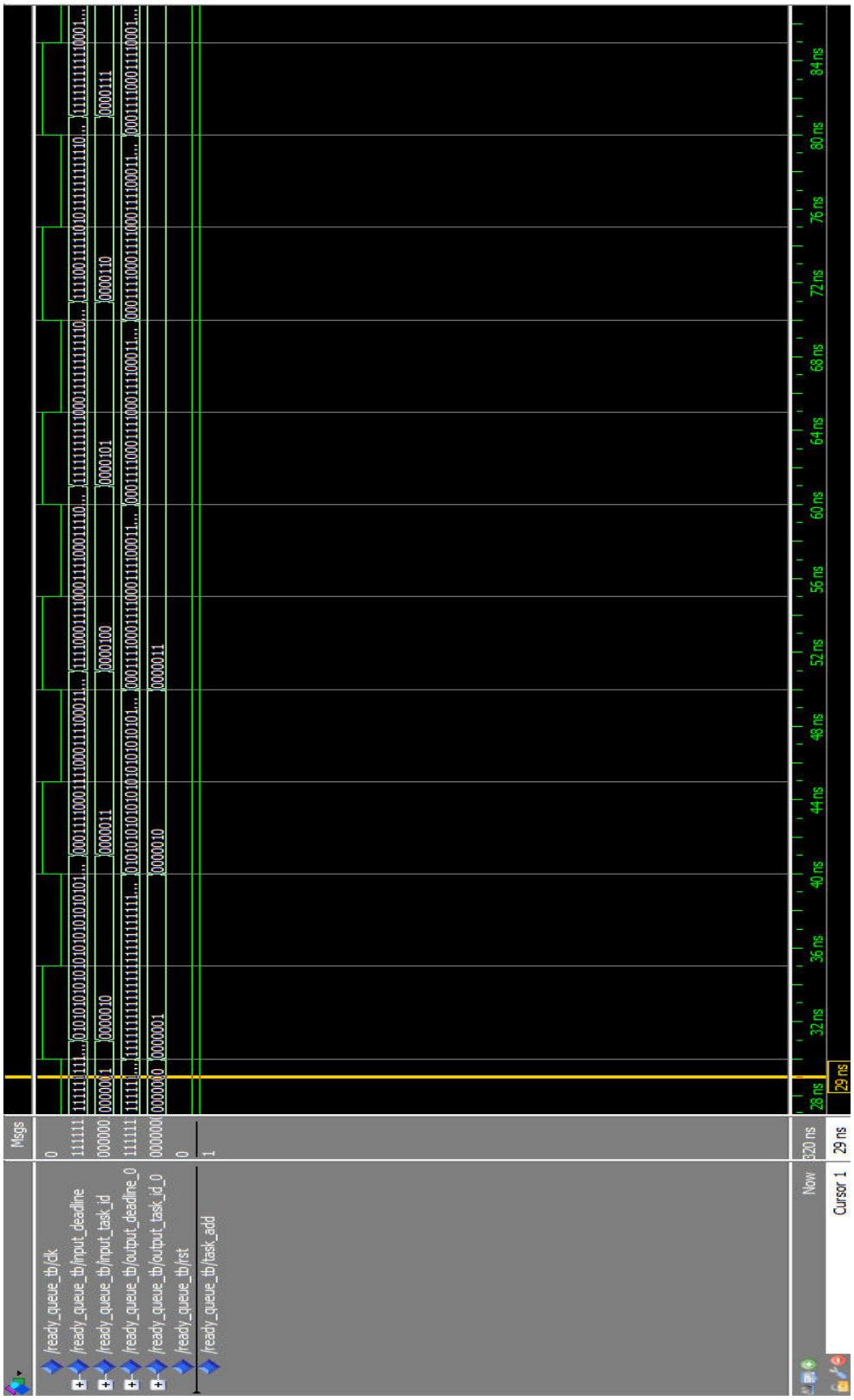
Obr. 3.12: Simulácia bunky posuvných registrov

Ako ďalšie sme verifikovali správne fungovanie zoznamu naplánovaných úloh, čiže hardvérové zoraďovanie úloh na základe systolického poľa obsahujúceho posuvné registre. Testovacie signály generované testbench-om môžeme opäť vidieť prostredníctvom grafu na obrázku 3.13.

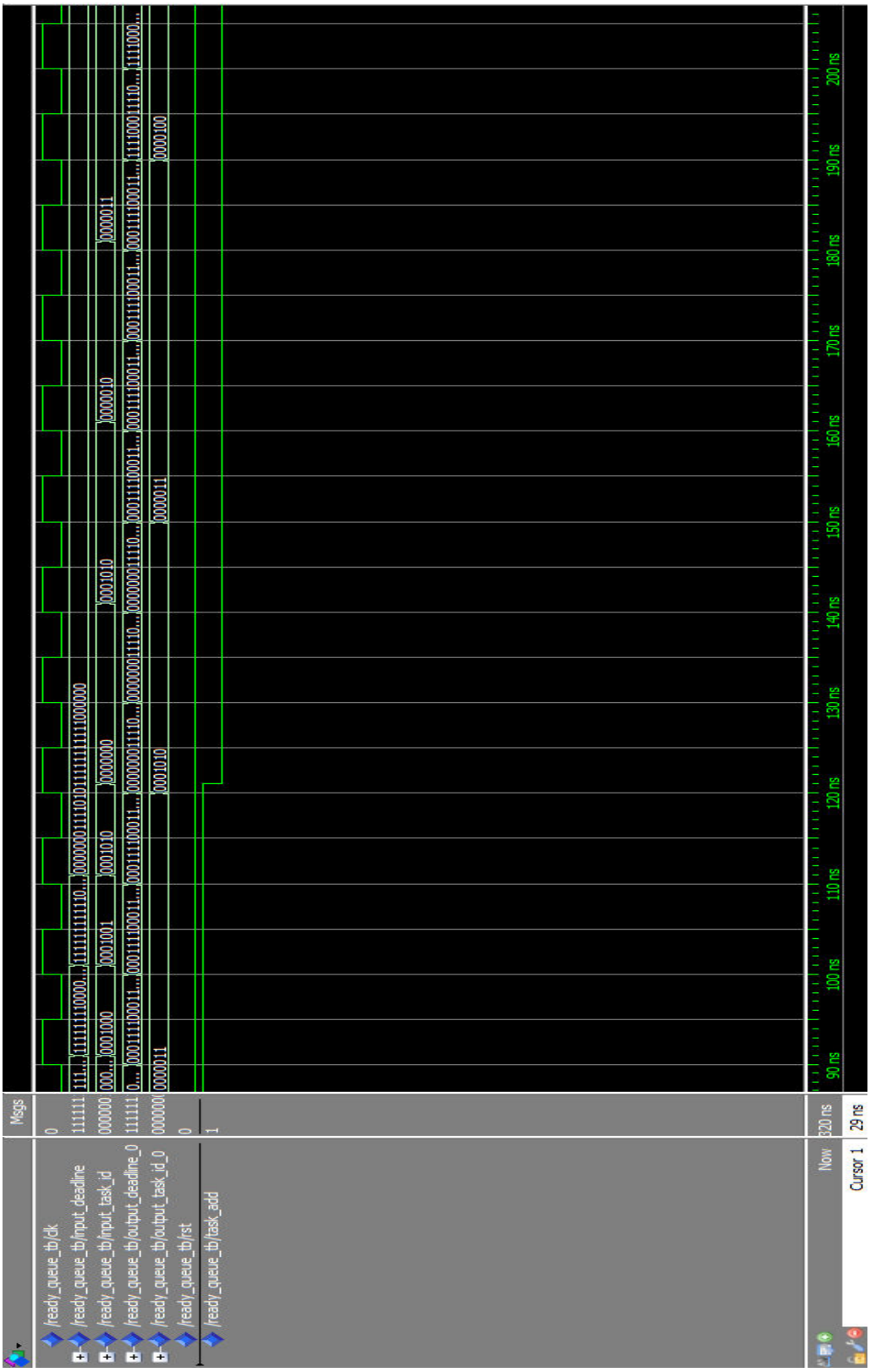


Obr. 3.13: Testovacie hodnoty pre zoraďovanie úloh

Nasledujú obrázky 3.14 a 3.15, ktoré znázorňujú priebehy simulácie hardvérového zoraďovania úloh. Prvý obrázok nám ukazuje správne fungovanie pridávania nových úloh do systému, zatiaľ čo druhý obrázok predstavuje ukážku mazania úloh podľa *ID*. Na pridanie úlohy potrebujeme vždy práve 1 impulz hodinového signálu, avšak na zmazanie úlohy potrebujeme 2 hodinové impulzy. Počas prvého impulzu prebieha mazanie danej úlohy a počas druhého impulzu sa vyprázdnia registre medzi prúdovými blokmi.



Obr. 3.14: Simulácia pridávania úloh do zoznamu naplánovaných úloh



Obr. 3.15: Simulácia mazania úloh zo zoznamu naplánovaných úloh

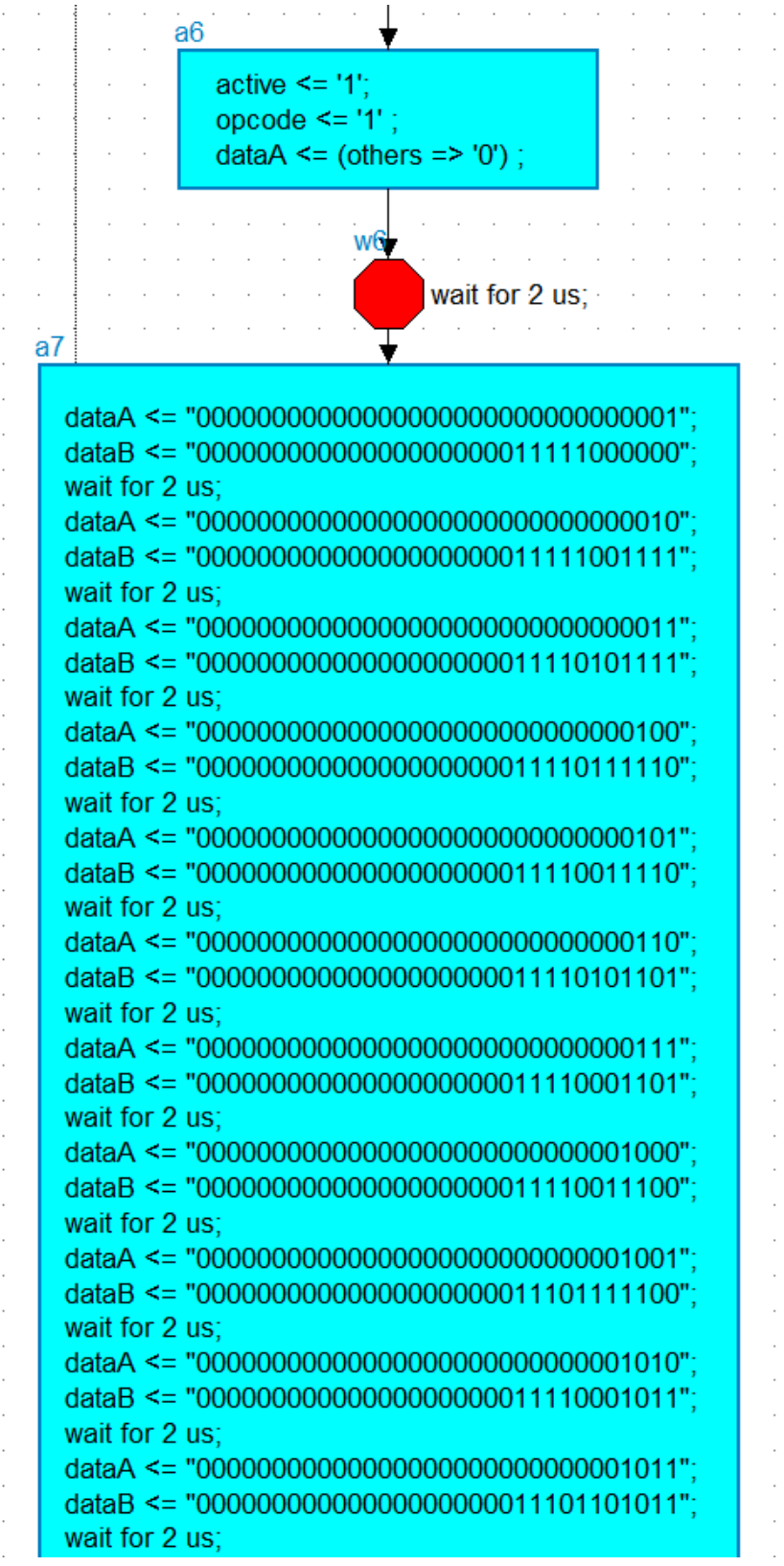
3.3.2 Simulácia koprocesora

Na tieto simulácie sme použili periódu hodín s dĺžkou 1 us, ktorá reprezentuje frekvenciu systému 1 MHz.

Verifikácia koprocesora spočíva v používaní inštrukcií koprocesora združených do skupín, ktoré realizujú nejaký scenár na vyššej úrovni abstrakcie. Rozhodli sme sa pre takúto postupnosť scenárov:

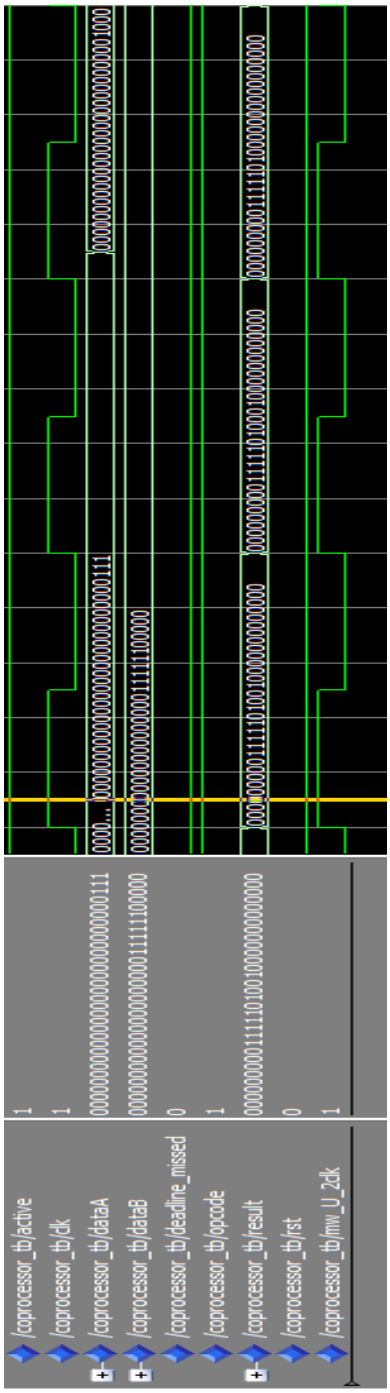
1. Naplnenie koprocesora všetkými úlohami, pričom každá úloha sa má vložiť na koniec zoznamu úloh. Toto sme zabezpečili sériou inštrukcií *task_schedule*, pričom každá používa rovnakú hodnotu *deadline*.
2. Vyprázdnenie koprocesora. Toto sme zabezpečili sériou inštrukcií *task_kill*, pričom *ID* postupne rástlo od hodnoty 0 do 31.
3. Naplnenie koprocesora všetkými úlohami, pričom každá druhá úloha sa má vložiť na začiatok zoznamu úloh a každá druhá úloha sa má vložiť na druhé miesto v zozname úloh. Toto sme zabezpečili sériou inštrukcií *task_schedule*, pričom *ID* postupne striedavo raz kleslo a raz zase stúplo.
4. Vyprázdnenie koprocesora rovnakým spôsobom ako v prípade bodu 2.
5. Naplnenie koprocesora všetkými úlohami, pričom každá úloha sa má vložiť na začiatok zoznamu úloh. Toto sme zabezpečili sériou inštrukcií *task_schedule,* pričom každá úloha má menší *deadline* ako predchádzajúca úloha.

Na overenie, či bol koprocesor navrhnutý správne a či sa správa podľa očakávaní, by mala vyššie popísaná postupnosť stačiť. Na obrázku 3.16 sme sa rozhodli zobraziť iba malú časť testovacích signálov z dôvodu nadmerného množstva riadkov, ktoré sa v danom testbench-i nachádzajú. Ako vidno z tohto obrázku, hodnoty *ID* aj *deadline* sa nachádzajú na spodných bitoch daných registrov, aby bola príprava dát čo najjednoduchšia a najrýchlejšia. Medzi každou inštrukciou sa čaká 2 mikrosekundy práve preto, lebo 1 takt procesora je v tomto prípade 1 mikrosekunda a inštrukcie koprocesora sa realizujú vždy po dobu dvoch taktov.



Obr. 3.16: Ukážka testovacích hodnôt pre koprocesor

Aj samotný priebeh simulácie bol príliš rozsiahly a preto sme vybrali len malý úsek z tejto simulácie ako ukážku. Jedná sa o vkladanie úloh v rámci prvého scenára z vybranej postupnosti scenárov. Ukážku simulácie môžeme vidieť na nasledovnom obrázku (obr. 3.17).



Obr. 3.17: Ukážka simulácie koprocesora

3.3.3 Softvérový plánovač úloh

Aby sme ešte viac overili, že výstup koprocesora je korektný, navrhli sme a implementovali aj softvérový plánovač úloh. Tento plánovač samozrejme taktiež realizuje algoritmus EDF rozšírený o schopnosť odstrániť ľubovoľnú úlohu zo zoznamu na základe jej *ID*.

Softvérový plánovač sme naprogramovali v jazyku C ako konzolovú aplikáciu, ktorá vypisuje výsledky plánovania na štandardný výstup konzoly. Tento plánovač využíva dátovú štruktúru spájaný zoznam na uchovávanie úloh v zoradenom zozname naplánovaných úloh.

Navrhli sme dva režimy, v ktorých vie pracovať tento plánovač úloh. Tieto režimy sú prepínané globálnou premennou EXECUTE, ktorá je buď true alebo false. Ak EXECUTE = true, tak plánovač úloh zároveň simuluje vykonávanie úloh a odstráni úlohu len vtedy, keď sa daná úloha dokončí alebo keď sa prekročí *deadline* danej úlohy. Ak EXECUTE = false, tak plánovač odstraňuje úlohy rovnakým postupom ako je postup v podkapitole 3.3.2 Simulácia koprocesora.

Keďže sme použili pre softvérový plánovač úloh rovnakú postupnosť inštrukcií ako pre simuláciu koprocesora, tak by sa mali výstupy oboch plánovačov zhodovať v každom čase. Tento jav sme odpozorovali na základe výstupných signálov simulácie koprocesora v ModelSim-e a výstupov konzolovej aplikácie softvérového plánovača (obr. 3.18). Týmto spôsobom sme overili, že navrhnutý hardvérový plánovač úloh má identické správanie ako softvérový plánovač úloh.

Výstup konzolovej aplikácie je interpretovaný nasledovným spôsobom. Skratka D vyjadruje *deadline*. Skratka EX vyjadruje „execution time“, čiže čas, koľko sa vykonáva daná úloha. Postupným vykonávaním úlohy sa zmenšuje EX až k hodnote 0, kedy je táto úloha dokončená. Vždy po vykonaní vloženia alebo zmazania úlohy sa zároveň vypíše výsledok „result“, čiže úloha, ktorá sa má vykonávať. Táto úloha je charakterizovaná jej ID, zostávajúcou dobou odozvy a zostávajúcou dobou vykonania.

```
>STARTING WITH TASK 0
ADDING TASK with ID: 0          D: 2016          EX: 127
RESULT is  ID: 0        D: 2014          EX: 125
ADDING TASK with ID: 1          D: 2016          EX: 127
RESULT is  ID: 0        D: 2012          EX: 123
ADDING TASK with ID: 2          D: 2016          EX: 127
RESULT is  ID: 0        D: 2010          EX: 121
ADDING TASK with ID: 3          D: 2016          EX: 127
RESULT is  ID: 0        D: 2008          EX: 119
ADDING TASK with ID: 4          D: 2016          EX: 127
RESULT is  ID: 0        D: 2006          EX: 117
ADDING TASK with ID: 5          D: 2016          EX: 127
RESULT is  ID: 0        D: 2004          EX: 115
ADDING TASK with ID: 6          D: 2016          EX: 127
RESULT is  ID: 0        D: 2002          EX: 113
ADDING TASK with ID: 7          D: 2016          EX: 127
RESULT is  ID: 0        D: 2000          EX: 111
ADDING TASK with ID: 8          D: 2016          EX: 127
RESULT is  ID: 0        D: 1998          EX: 109
ADDING TASK with ID: 9          D: 2016          EX: 127
RESULT is  ID: 0        D: 1996          EX: 107
ADDING TASK with ID: 10         D: 2016          EX: 127
RESULT is  ID: 0        D: 1994          EX: 105
ADDING TASK with ID: 11         D: 2016          EX: 127
RESULT is  ID: 0        D: 1992          EX: 103
ADDING TASK with ID: 12         D: 2016          EX: 127
RESULT is  ID: 0        D: 1990          EX: 101
ADDING TASK with ID: 13         D: 2016          EX: 127
RESULT is  ID: 0        D: 1988          EX: 99
ADDING TASK with ID: 14         D: 2016          EX: 127
RESULT is  ID: 0        D: 1986          EX: 97
ADDING TASK with ID: 15         D: 2016          EX: 127
RESULT is  ID: 0        D: 1984          EX: 95
ADDING TASK with ID: 16         D: 2016          EX: 127
RESULT is  ID: 0        D: 1982          EX: 93
ADDING TASK with ID: 17         D: 2016          EX: 127
RESULT is  ID: 0        D: 1980          EX: 91
ADDING TASK with ID: 18         D: 2016          EX: 127
RESULT is  ID: 0        D: 1978          EX: 89
ADDING TASK with ID: 19         D: 2016          EX: 127
RESULT is  ID: 0        D: 1976          EX: 87
ADDING TASK with ID: 20         D: 2016          EX: 127
RESULT is  ID: 0        D: 1974          EX: 85
ADDING TASK with ID: 21         D: 2016          EX: 127
RESULT is  ID: 0        D: 1972          EX: 83
ADDING TASK with ID: 22         D: 2016          EX: 127
RESULT is  ID: 0        D: 1970          EX: 81
ADDING TASK with ID: 23         D: 2016          EX: 127
RESULT is  ID: 0        D: 1968          EX: 79
ADDING TASK with ID: 24         D: 2016          EX: 127
RESULT is  ID: 0        D: 1966          EX: 77
ADDING TASK with ID: 25         D: 2016          EX: 127
RESULT is  ID: 0        D: 1964          EX: 75
ADDING TASK with ID: 26         D: 2016          EX: 127
RESULT is  ID: 0        D: 1962          EX: 73
ADDING TASK with ID: 27         D: 2016          EX: 127
RESULT is  ID: 0        D: 1960          EX: 71
ADDING TASK with ID: 28         D: 2016          EX: 127
RESULT is  ID: 0        D: 1958          EX: 69
ADDING TASK with ID: 29         D: 2016          EX: 127
RESULT is  ID: 0        D: 1956          EX: 67
ADDING TASK with ID: 30         D: 2016          EX: 127
RESULT is  ID: 0        D: 1954          EX: 65
ADDING TASK with ID: 31         D: 2016          EX: 127
RESULT is  ID: 0        D: 1952          EX: 63
>PRESS ENTER TO CONTINUE
>_
```

Obr. 3.18: Výstup softvérového plánovača úloh

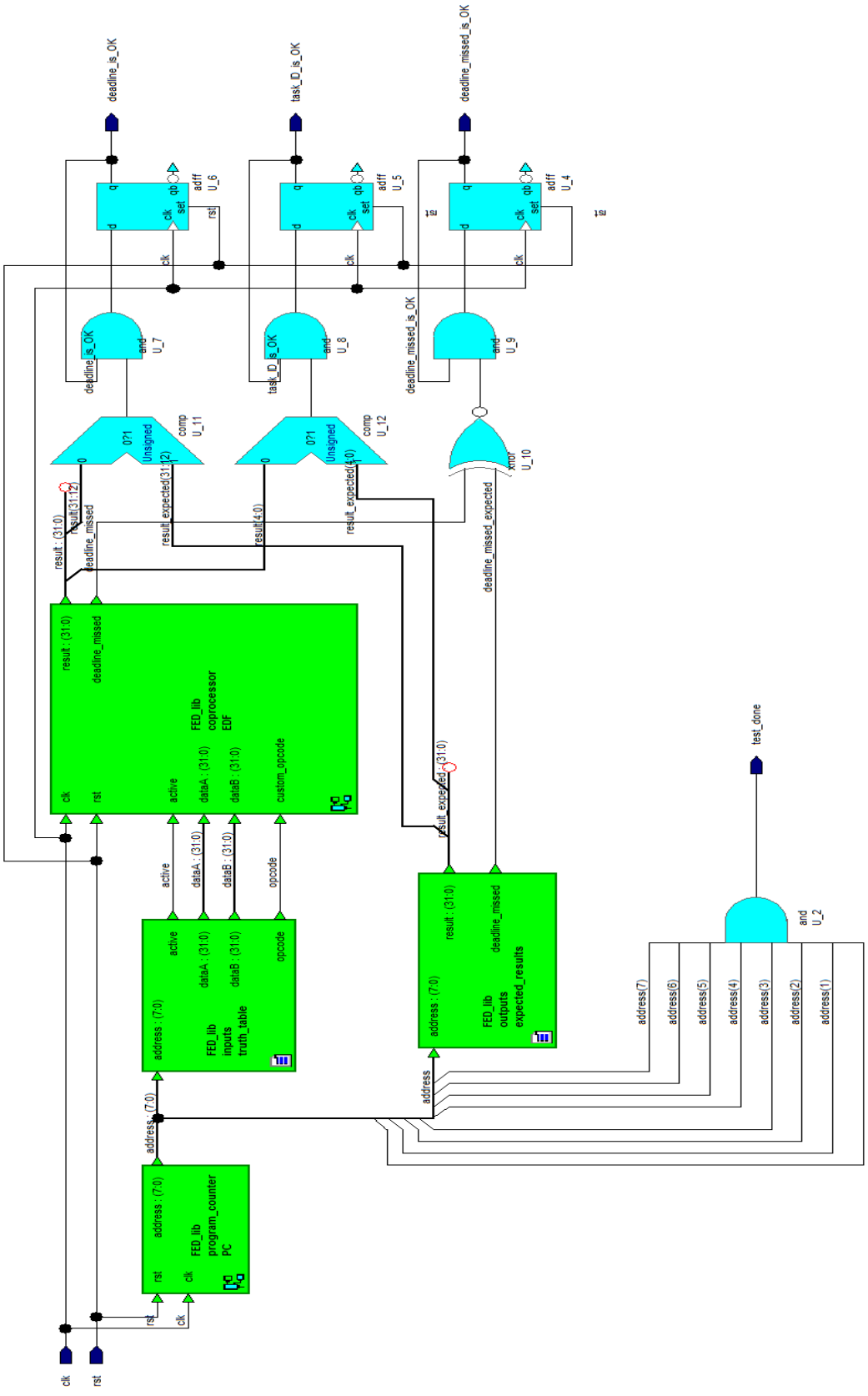
Rýchlosť oboch plánovačov sme síce neporovnali experimentálne, ale vykonali sme analytické porovnanie nasledovným spôsobom. Keďže softvérový plánovač úloh používa spájaný zoznam, je zrejmé, že asymptotická zložitosť takéhoto plánovača je O(N), kde N predstavuje počet naplánovaných úloh. Naproti tomu je rýchlosť hardvérového plánovača nezávislá od počtu naplánovaných úloh, ba dokonca je nezávislá aj od maximálneho množstva úloh v systéme (kapacity plánovača), pretože inštrukcie hardvérového plánovača úloh trvajú vždy presne 2 takty procesora.

3.3.4 Funkcionálny BIST

Pre navrhnutý koprocesor sme tiež navrhli funkcionálny spôsob vstavaného samočinného testovania, ktorý sme tiež odsimulovali v nástroji ModelSim. Na tieto simulácie sme taktiež použili periódu hodín s dĺžkou 1 us, ktorá reprezentuje frekvenciu systému 1 MHz.

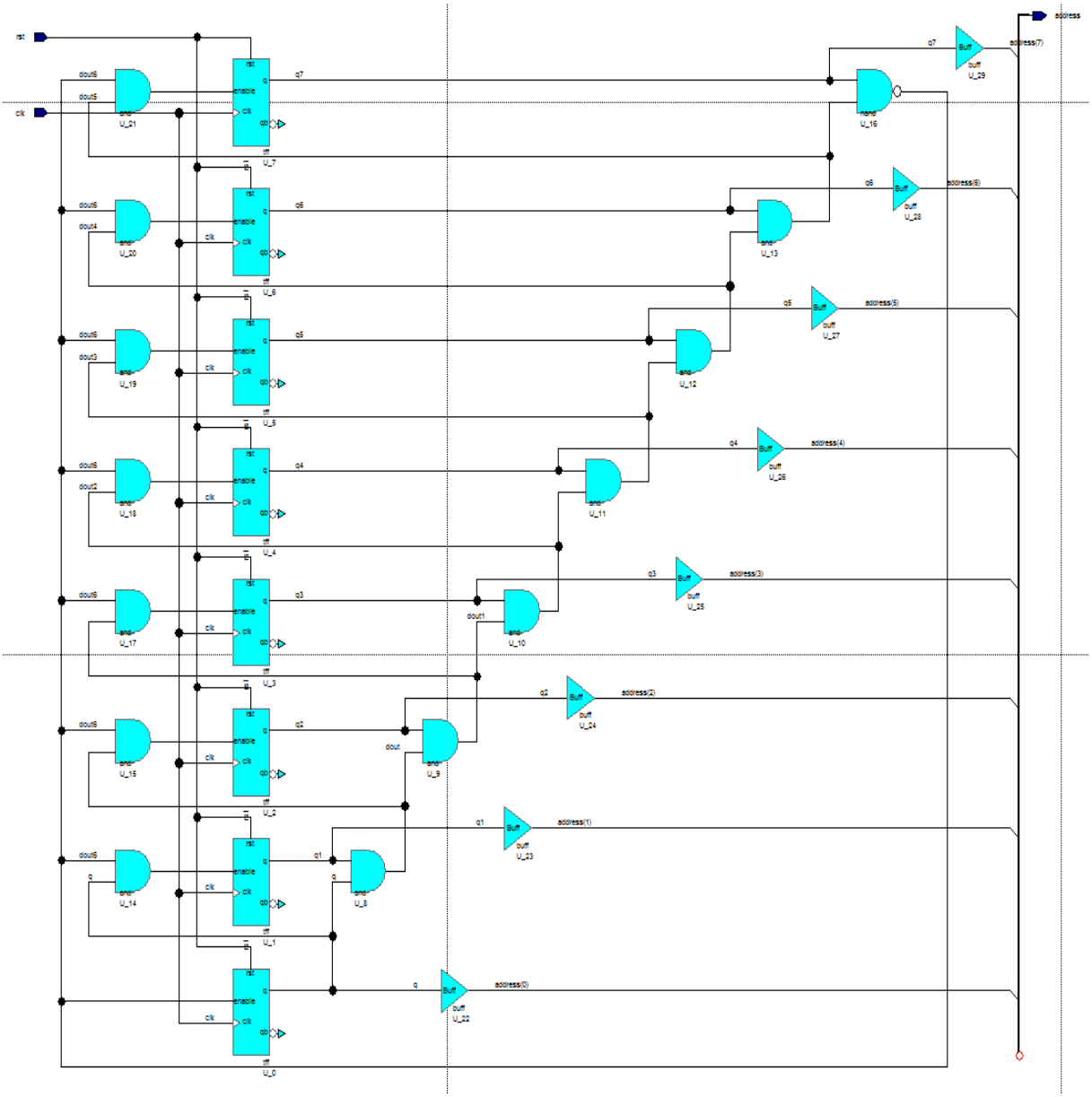
Zatiaľ čo štandardný BIST je založený na štruktúrnom testovaní obvodu, náš BIST používa funkcionálne testovanie, ktoré je založené na inštrukciach koprocesora, rovnako ako v predchádzajúcich simuláciach. Takýto BIST nám vo svojej podstate simuluje použitie nejakého procesora a operačného systému, vďaka čomu nie je potrebné verifikovať daný koprocesor v spojení s nejakým konkrétnym CPU a OS.

Náš funkcionálny BIST používa jednoduché počítadlo adries, pri ktorom sme sa inšpirovali od počítadla „program counter“ nachádzajúcom sa v každom CPU. Toto počítadlo postupne generuje adresy od hodnoty 0 a vyššie. Na základe použitých adries generuje jeden kombinačný obvod vstupné dáta pre koprocesor a jeden kombinačný obvod očakávané výstupné dáta koprocesora. Takéto dve kombinačné logiky je ideálne opísať iba formou správania (napr. pravdivostnou tabuľkou alebo procesom) a nechať na kompilátore, aby vykonal logickú syntézu daného opisu. Opis správania je v tomto prípade vhodnejší z toho dôvodu, že generovaných inštrukcií môže byť pomerne mnoho a takáto forma opisu je ľahko modifikovateľná v prípade, že sa rozhodneme použiť inú postupnosť inštrukcií. Použité kombinačné obvody môžu byť relatívne efektívne realizované pomocou multiplexorových stromov alebo pomocou pamäte ROM. Na koniec iba porovnáme výstupy vygenerované koprocesorom s očakávanými výstupmi, pričom nezhoda v akomkoľvek čase zmení výstup BIST obvodu z logickej jednotky na nulu. Celkovo obsahuje náš BIST 4 výstupné bity. Jeden informuje o tom, kedy je koniec testovania, druhý informuje o tom, či bol počas celého testovania v poriadku výstup *deadline_missed*, tretí informuje o tom, či bol počas celého testovania v poriadku výstup *deadline* a štvrtý informuje o tom, či bol počas celého tetovania v poriadku výstup *ID*. Štruktúru BIST obvodu môžeme vidieť na obrázku 3.19.



Obr. 3.19: Štruktúra funkcionálneho BIST-u

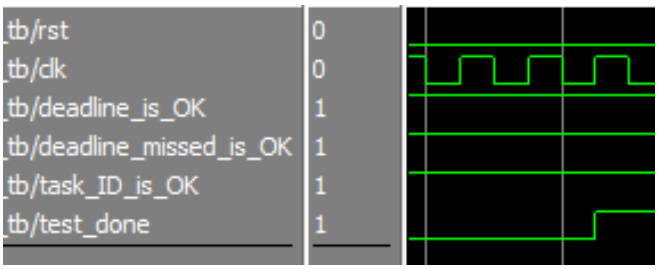
Len pre zaujímavosť môžeme vidieť na obrázku 3.20 štruktúru generátora adries.



Obr. 3.20: Štruktúra generátora adries

Táto štruktúra obsahuje T preklápacie obvody, ktoré sú ideálne pre realizáciu počítadiel. Každý T preklápací obvod obsahuje iba jeden D preklápací obvod a jeden multiplexor. Logické hradlá AND nachádzajúce sa naľavo od T preklápacích obvodov sú pridané do štruktúry kvôli tomu, aby sa počítadlo adries automaticky zastavilo na poslednej adrese, čiže adrese, ktorá obsahuje samé jednotky.

Úspešný koniec simulácie testovania koprocesora funkcionálnym BIST-om môžeme vidieť na nasledovnom obrázku (obr. 3.21). Úspešný je samozrejme preto, lebo všetky tri signály *task_ID_is_OK*, *deadline_is_OK* a *deadline_missed_is_OK* sú nastavené na hodnotu log. 1 na konci simuácie. Koniec simulácie poznáme na základe signálu *test_done*. Ak by sa v ľubovoľnom čase počas celého testovania nezhodoval niektorý výstup z koprocesora s očakávaným výstupom, tak by sa od toho momentu držala v niektorom z vyššie spomínaných výstupných signálov BIST-u hodnota log. 0. Preto nám stačí sledovať len koniec testovania.



Obr. 3.21: Koniec simulácie funkcionálneho BIST testovania

3.3.5 Syntéza a implementácia na platforme FPGA

Navrhnutý koprocesor sme syntetizovali a implementovali na FPGA, pričom boli použité tieto parametre koprocesora:

- Kapacita koprocesora: 32 úloh
- Počet prúdových stupňov: 4
- Šírka deadline: 20 bitov

Syntéza navrhnutého koprocesora bola vykonaná pre dve konkrétne FPGA zariadenia, ktorými sú Xilinx Spartan 3A a Altera Cyclone II.

Výsledok syntézy pre Altera Cyclone II bol získaný z programu Altera Quartus II a tento výsledok môžeme vidieť na obrázku 3.22. Na tomto obrázku si môžeme všimnúť, že počet použitých registrov presne zodpovedá teoreticky vypočítanému počtu D preklápacích obvodov, ktorý sme analyticky vypočítali v kapitole 3.2.7 Teoretická spotreba plochy na čipe.

Flow Status	Successful - Fri May 08 12:03:30 2015
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	druhy_projekt
Top-level Entity Name	coprocessor
Family	Cyclone II
Device	EP2C70F896C6
Timing Models	Final
Total logic elements	4,120 / 68,416 (6 %)
Total combinational functions	4,119 / 68,416 (6 %)
Dedicated logic registers	957 / 68,416 (1 %)
Total registers	957
Total pins	101 / 622 (16 %)
Total virtual pins	0
Total memory bits	0 / 1,152,000 (0 %)
Embedded Multiplier 9-bit elements	0 / 300 (0 %)
Total PLLs	0 / 4 (0 %)

Obr. 3.22: Výsledok syntézy koprocesora pre FPGA Altera Cyclone II

Na nasledovnom obrázku (obr. 3.23) môžeme vidieť výsledok syntézy pre Xilinx Spartan 3, ktorú sme vykonali v programe Xilinx ISE Design Suite.

coprocessor Project Status (05/02/2015 - 15:13:42)			
Project File:	Diplomovka.xise	**Parser Errors:**	No Err
Module Name:	coprocessor	**Implementation State:**	Placed
Target Device:	xc3s400a-5ft256	**•Errors:**	No Err
Product Version:	ISE 14.7	**•Warnings:**	70 Wa
Design Goal:	Balanced	**•Routing Results:**	All Sig
Design Strategy:	Xilinx Default (unlocked)	**•Timing Constraints:**	All Co
Environment:	System Settings	**•Final Timing Score:**	0 (Tim

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	967	7,168	13%
Number of 4 input LUTs	4,355	7,168	60%
Number of occupied Slices	2,277	3,584	63%
Number of Slices containing only related logic	2,277	2,277	100%
Number of Slices containing unrelated logic	0	2,277	0%
Total Number of 4 input LUTs	4,387	7,168	61%
Number used as logic	4,355		
Number used as a route-thru	32		
Number of bonded IOBs	62	195	31%
Number of BUFGMUXs	1	24	4%
Average Fanout of Non-Clock Nets	3.94		

Obr. 3.23: Výsledok syntézy koprocesora pre FPGA Xilinx Spartan 3

Funkcionálny BIST, ktorý sme navrhli a verifikovali simuláciou v predchádzajúcej podkapitole, nám umožňuje relatívne veľmi jednoduché overenie správneho fungovania nášho koprocesora implementovaného na platforme FPGA. Jediné, čo bolo potrebné doriešiť, bolo namapovanie vstupov a výstupov BIST obvodu na piny FPGA čipu a nakonfigurovanie FPGA zariadenia .bit súborom prostredníctvom USB. Výstupné signály BIST obvodu boli namapované na piny FPGA čipu tak, aby boli tieto signály vyvedené na štyri LED diódy. Vstupné signály pre BIST obvodu sú iba resetovací signál a hodinový signál. Resetovací signál bol namapovaný tak, aby reagoval na resetovacie tlačidlo. Týmto spôsobom sme otestovali navrhnutý koprocesor na FPGA.

Keďže sme náš koprocesor rozšírili o navrhnutý funkcionálny BIST, považujeme za zaujímavé zistiť, ako sa zvýšila spotreba plochy na čipe. Túto informáciu môžeme vidieť v nasledovných dvoch tabuľkách (tabuľka 3 a tabuľka 4).

	Iba koprocesor	Koprocesor + BIST	Relatívne navýšenie
Počet logických elementov	4120	4260	+ 3%
Počet kombinačných funkcií	4119	4256	+ 3%
Počet registrov	957	950	- 1%

Tabuľka 3: Cena za BIST na FPGA Altera Cyclone II

	Iba koprocesor	Koprocesor + BIST	Relatívne navýšenie
Počet LUT	4355	4443	+ 2%
Počet prepojov	2277	2353	+ 3%
Počet registrov	967	996	+ 3%

Tabuľka 4: Cena za BIST na FPGA Xilinx Spartan 3

4 Zhodnotenie

Boli analyzované všetky potrebné informácie o operačných systémoch, obzvlášť plánovacie algoritmy pre systémy reálneho času, možností ich hardvérovej realizácie a akcelerácie, a existujúce hardvérové plánovače úloh určené pre systémy reálneho času. Na základe vykonanej analýzy sme najprv navrhli nový plánovací algoritmus FED, ktorý sa líši od ostatných existujúcich algoritmov tým, že podporuje spoľahlivé kombinovanie plánovania non-RT, firm RT a hard RT úloh. Neskôr sme ale zvážili výhody a nevýhody algoritmu FED voči existujúcemu algoritmu EDF a rozhodli sme sa radšej použiť EDF, ktorý vyžaduje omnoho menej logiky na hardvérovú realizáciu.

V rámci diplomovej práce sme navrhli efektívnu a škálovateľnú architektúru zoraďovania úloh. Táto architektúra je založená na posuvných registroch so spoločným paralelným vstupom a na technike prúdového spracovania inštrukcií známej pod pojmom pipelining. Navrhnutú architektúru sme odsimulovali, čím sme verifikovali korektnosť návrhu.

Následne sme navrhli hardvérový plánovač úloh vo forme koprocesora, ktorý realizuje vybraný algoritmus EDF rozšírený o schopnosť odstrániť na požiadanie ľubovoľnú úlohu v systéme na základe jej ID. Tento koprocesor používa štandardné rozhranie založené na dvoch vstupných registroch a jednom výstupnom registri. Tento koprocesor má dve inštrukcie, pričom realizácia trvá vždy 2 takty procesora bez ohľadu na počet naplánovaných úloh a maximálny počet úloh v koprocesore. Vďaka tomu, že inštrukcie sa vykonávajú na dva takty, je hardvérové plánovanie výrazne rýchlejšie ako softvérové plánovanie úloh. A nielen že je rýchlejšie, ale je dokonca deterministické, pretože má konštantnú časovú zložitosť.

Na základe navrhnutého koprocesora sme analyzovali jeho spotrebu plochy na čipe na úrovni logických hradiel a základných aritmeticko-logických operácií. Táto spotreba najviac závisí od maximálneho počtu úloh v systéme a presnosti počítania času pre deadline úloh. V malej miere závisia nároky na plochu na čipe aj od toho, koľko prúdových stupňov použijeme.

Na verifikáciu koprocesora sme použili okrem simulácií v ModelSim-e aj softvérovú implementáciu plánovača úloh, funkcionálny BIST a implementáciu na dvoch platformách FPGA. Softvérový plánovač úloh v podobe konzolovej aplikácie napísanej v jazyku C bol založený na spájanom zozname. Konzolový výstup tejto aplikácie bol porovnaný s výstupmi simulácie koprocesora, čím bolo ešte lepšie overené, že navrhnutý koprocesor funguje správne. Okrem toho sme overili koprocesor aj implementáciou na platforme FPGA Altera

a FPGA Xilinx. Na to, aby sme nemuseli používať nejaký existujúci procesor a operačný systém, sme radšej vyvinuli funkcionálny BIST, ktorý nahrádza operačný systém tým, že ho simuluje. Tento BIST generuje pre koprocesor inštrukcie a očakávané výstupy koprocesora, ktoré porovnáva s reálnymi výstupmi koprocesora. Na základe týchto porovnaní svietia LED diódy na FPGA. Nami navrhnutý BIST je vhodný nielen na verifikáciu, ale aj na testovanie koprocesora priebežne počas fungovania celého systému. Z hľadiska pridaných nárokov na plochu na čipe predstavuje tento BIST nárast približne o 3%.

Najväčším prínosom tejto práce je sprístupnenie algoritmu EDF pre praktické využitie, nakoľko softvérové realizácie tohto algoritmu majú neprijateľnú časovú zložitosť. Systémy reálneho času potrebujú používať plánovanie úloh s konštantnou časovou zložitosťou a preto sa algoritmus EDF nepoužíva. Namiesto neho používajú operačné systémy reálneho času algoritmy založené na prioritách úloh. Je nutné zdôrazniť, že tieto účinnosť týchto algoritmov vo veľkej miere závisí od správneho pridelenia priorít jednotlivým úlohám. Naproti tomu algoritmus EDF eliminuje problém prideľovania priorít, pretože je založený výhradne na deadline úloh. Ak máme systém, ktorý obsahuje iba hard RT úlohy, tak je náš koprocesor ideálny bez akýchkoľvek rozšírení. Ak ale chceme použiť algoritmus EDF aj pre iné typy úloh, je vhodnejšie rozšíriť tento koprocesor o ďalšiu funkcionalitu, ktorá môže byť realizovaná softvérovo. Práve pre rozšíriteľnosť o novú funkcionalitu bol koprocesor navrhnutý tak, aby sme mohli odstrániť ľubovoľnú úlohu zo zoznamu naplánovaných úloh.

5 Technická dokumentácia

Táto kapitola obsahuje návod na použitie koprocesora a návod na otestovanie koprocesora.

5.1 Návod na použitie koprocesora

Na použitie koprocesora je potrebné v prvom rade skopírovať .hdl súbory do priečinka svojho HDL projektu. Následne stačí iba vložiť entitu *coprocessor* ako komponent vnútri niektorej vlastnej entity.

Z hľadiska architektúry procesora, ktorý má byť rozšírený o tento koprocesor, predstavuje koprocesor rozšírenie aritmeticko-logickej jednotky procesora. Okrem toho však treba rozšíriť ešte dekódovaciu časť procesora, aby boli rozoznávané inštrukcie určené pre koprocesor.

Ako posledné treba ešte zohľadniť šírky jednotlivých signálov ako rozhrania koprocesora, tak aj vnútri koprocesora, čiže počty bitov. Napríklad počet bitov použitých na reprezentáciu *ID* úlohy priamo súvisí s kapacitou koprocesora z hľadiska maximálneho počtu úloh. Počet bitov signálov *dataA*, *dataB* a *result* by mal byť zhodný s tým, koľko bitov majú registre daného CPU. Počet bitov použitých na *deadline* je 20, avšak je vhodné zvážiť aj iný počet bitov vzhľadom na konkrétnu aplikáciu a požiadavky na presnosť časovania systému.

5.2 Návod na otestovanie koprocesora

Pre otestovanie koprocesora na FPGA Xilinx Spartan 3 je potrebné použiť súbor *testing_environment.bit*. Tento súbor stačí nahrať do FPGA cez USB kábel pomocou programu na to určeného, napríklad AvProg. Po nahratí daného súboru by mali svietiť všetky 4 LED diódy, ktoré sa nachádzajú na danom FPGA. Stlačením resetovacieho tlačidla je krajná LED dióda zhasnutá, avšak iba po dobu držania tlačidla. Táto dióda reprezentuje výstupný signál *test_done*.

Pre otestovanie koprocesora na FPGA Altera Cyclone II platí skoro to isté ako v prípade vyššie. Rozdiel je len v tom, že namiesto súboru *testing_environment.bit* treba použiť súbor *druhy_projekt.sof*.

6 Zoznam použitej literatúry

[BLOOM, 2010] BLOOM, G., PARMER, G., NARAHARI, B., SIMHA, R.: Real-Time Scheduling with Hardware Data Structures, 2010.

[BUTTAZZO, 2011] BUTTAZZO, G. C.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2011, ISBN 9781461406754, 9781461406761.

[CHANDRA, 2010] CHANDRA, R., SINNEN O.: Improving Application Performance with Hardware Data Structures, 2010, ISSN 11783680.

[E-UIVERSITY, 2013] Wisdom IT Services India Pvt. Ltd http://e-university.wisdomjobs.com/data-structures/chapter-1177-290/heaps.html.

[FERREIRA, 2010] FERREIRA, C., OLIVEIRA, A. S. R.: Hardware Co-Processor for the OReK Real-Time Executive, 2010.

[FREEBSD, 2013] The FreeBSD Project, 2013. http://www.freebsd.org/.

[FREERTOS, 2013] The FreeRTOS Project, 2013. http://www.freertos.org/.

[GOKHALE, 2010] GOKHALE, M. B.; GRAHAM, P. S.: Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays. Springer Publishing Company, Incorporated, prve vydanie, 2010, ISBN 1441938656, 9781441938657.

[HAUCK, 2008] HAUCK, S., DEHON, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation, 2008.

[HENNESSY, 2011] HENNESSY, J. L., PATTERSON, D. A.: Computer Architecture, Fifth Edition: A Quantitative Approach, 2011, ISBN 9780123838728.

[KOHOUT, 2002] KOHOUT, P.: Hardware Support for Real-Time Operating Systems, 2002.

[LABROSSE, 2007] LABROSSE, J. J., GANSSLE, J., OSHANA, R., e. a.: Embedded Software: Know It All (Newnes Know It All). Newnes, 2007, ISBN 0750685832.

[LANGE, 2012] LANGE, A. B., ANDERSEN, K. H., SCHULTZ, U. P., SORENSEN, A. S.: HartOS - a Hardware Implemented RTOS for Hard Real-time Applications, 2012, s. 207-213.

[LYSECKY, 2005] LYSECKY, R., VAHID, F.: A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning, 2005.

[MOON, 2000] MOON, S. W., SHIN, K. G., REXFORD, J.: Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches, 2000, ISSN 00189340.

[MOORE, 1995] MOORE, S. W., GRAHAM B. T.: Tagged Up/Down Sorter – A Hardware Priority Queue, 1995.

[NUTTX, 2015] NuttX Real-Time Operating System, 2015. http://www.nuttx.org

[ONG, 2013] ONG, S. E., LEE S. C.: SEOS: Hardware Implementation of Real-Time Operating System for Adaptability, 2013, ISBN 9781479927951.

[ROSINGER, 2004] ROSINGER, H.: Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel, 2004.

[RTWIKI, 2008] Real-Time Linux Wiki, 2008. https://rt.wiki.kernel.org

[STEPHENSON, 2005] STEPHENSON, J.: Design Guidelines for Optimal Results in FPGAs, 2005.

[TANENBAUM, 2001] TANENBAUM, A. S.: Modern Operating Systems (2nd Edition) (GOAL Series). Prentice Hall, 2001.

[VAHID, 1999] VAHID, F., GIVARGIS, T.: Embedded System Design: A Unified Hardware/Software Approach, 1999.

[VOJTKO, 2012] VOJTKO, M.: Modulárny operačný systém pre vnorené systémy. Bratislava: FIIT STU, 2012, Diplomová práca, pp. 3 – 65.

[WOODHULL, 2006] TANENBAUM, A. S., WOODHULL, A. S.: Operating Systems Design and Implementation (3rd Edition). Prentice Hall, 2006, ISBN 0131429388.

Príloha A Obsah elektronického média

/FPGA Altera/	– priečinok obsahujúci zosyntetizovaný projekt pre FPGA Altera Cyclone II
/FPGA Xilinx/	– priečinok obsahujúci zosyntetizovaný projekt pre FPGA Xilinx Spartan 3
/HDL Projekt/	– priečinok obsahujúci VHDL súbory a projekt pre nástroj *Mentor Graphics HDL Designer*
/Literatúra/	– priečinok obsahujúci vybrané dokumenty použitej literatúry
/SW/	– priečinok obsahujúci softvérový plánovač úloh
/Annotation.pdf	– anotácia v anglickom jazyku
/Anotácia.pdf	– anotácia v slovenskom jazyku
/Diplomová práca.docx	– tento dokument vo formáte .docx
/Diplomová práca.pdf	– tento dokument vo formáte .pdf
/Návrh zadania.pdf	– návrh zadania diplomovej práce
/Obal práce.pdf	– obal diplomovej práce
/Obsah.txt	– obsah elektronického média
/Zadanie.pdf	– znenie zadania diplomovej práce

Príloha B Plán práce na diplomovom projekte 1

Každý bod zodpovedá časovému obdobiu v podobe približne jedného týždňa:

1. Štúdium relevantných materiálov, čítanie kníh, vedeckých článkov a ostatných študijných materiálov týkajúcich sa existujúcich operačných systémov a ich častí
2. Analýza problému správy úloh – z pohľadu tvorby a odstránenia úloh, a dát o úlohe
3. Analýza problému správy úloh – plánovanie úloh
4. Analýza problému správy úloh – komunikácia medzi úlohami
5. Analýza problému správy pamäte – algoritmy na výber voľného bloku pamäte
6. Analýza problému časovej a pamäťovej náročnosti možných riešení zadania a zložitosti hardvéru
7. Analýza programovateľných logických obvodov CPLD a FPGA
8. Analýza možností komunikácie obvodu s počítačom najmä s ohľadom na rýchlosť
9. Analýza existujúcich hardvérových riešení a ich zhodnotenie
10. Výber vhodného procesora, vhodného programovateľného obvodu a vhodných častí operačného systému, ktoré budú realizované hardvérovo
11. Špecifikácia požiadaviek na hardvérový akcelerátor
12. Hrubý návrh algoritmov realizovaných hardvérovo

Príloha C Plán práce na diplomovom projekte 2

Každý bod zodpovedá časovému obdobiu v podobe približne jedného týždňa:

1. Zapracovanie pripomienok vedúceho projektu, ktoré boli získané na konci semestra v rámci hodnotenia diplomového projektu 1
2. Výber zamerania plánovača úloh na určitú oblasť systémov
3. Špecifikácia funkcionálnych požiadaviek na plánovač úloh
4. Návrh nového plánovacieho algoritmu (1/2)
5. Návrh nového plánovacieho algoritmu (2/2)
6. Návrh architektúry hardvérového plánovača úloh (1/2)
7. Návrh architektúry hardvérového plánovača úloh (2/2)
8. Návrh hardvérovej štruktúry určenej pre zoraďovanie úloh (1/2)
9. Návrh hardvérovej štruktúry určenej pre zoraďovanie úloh (2/2)
10. Testovanie hardvérovej štruktúry určenej pre zoraďovanie úloh
11. Návrh štruktúry ostatných častí hardvérového plánovača úloh
12. Testovanie štruktúry ostatných častí hardvérového plánovača úloh

Príloha D Plán práce na diplomovom projekte 3

Každý bod zodpovedá časovému obdobiu v podobe približne jedného týždňa:

1. Dokončenie návrhu štruktúry celého hardvérového plánovača
2. Pridanie komunikačného rozhrania pre procesor
3. Testovanie celého systému pomocou simulácie
4. Implementácia navrhnutého systému na vybrané FPGA (1/2)
5. Implementácia navrhnutého systému na vybrané FPGA (2/2)
6. Testovanie implementovaného systému
7. Implementácia softvérového plánovača úloh
8. Testovanie softvérového plánovača úloh
9. Porovnávanie výsledkov a rýchlosti oboch riešení
10. Dokumentácia výsledkov, tvorba tabuliek a grafov
11. Finalizácia práce a zhodnotenie
12. Príprava prezentácie na obhajobu práce