Random Access Memory for a Multi-Processor System on Chip

QueenField

0. INTRODUCTION

0.0. DO-254

0.0.1. HARDWARE PLANNING PROCESS

0.0.1.1. Plan for Hardware Aspects of Certification

0.0.1.2. Hardware Design Plan

0.0.1.3. Hardware Validation Plan

0.0.1.4. Hardware Verification Plan

0.0.1.5. Hardware Configuration Management Plan

0.0.1.6. Hardware Process Assurance Plan

0.0.2. HARDWARE DESIGN PROCESS

0.0.2.1 Requirements Capture Process

0.0.2.2 Conceptual Design Process

0.0.2.3 Detailed Design Process

0.0.2.4 Implementation Process

0.0.2.5 Production Transition Process

0.0.3. VALIDATION AND VERIFICATION PROCESS

0.0.3.1 Validation Process

0.0.3.2 Verification Process

0.0.4. CONFIGURATION MANAGEMENT PROCESS

0.0.5. PROCESS ASSURANCE

0.0.6. CERTIFICATION LIAISON PROCESS

0.0.7. HARDWARE DESIGN LIFECYCLE DATA

0.0.7.1 Certification Authority

0.0.7.2 Certification Reviews

0.0.7.3 Scheduling of Reviews

0.0.8. ADDITIONAL CONSIDERATIONS

0.0.8.1 Previously Developed Hardware

0.0.8.2 Commercial Components Usage

0.1. BEST PRACTICES

0.1.1. HARDWARE

0.1.2. SOFTWARE

0.1. OPEN SOURCE PHILOSOPHY

For Windows Users!

1. Settings → Apps → Apps & features → Related settings, Programs and Features → Turn Windows features on or off → Windows Subsystem for Linux
2. Microsoft Store → INSTALL UBUNTU

type:

```
sudo apt update
sudo apt upgrade
```

0.2.1. Open Source Hardware

0.2.1.1. MSP430 Processing Unit

0.2.1.2. OpenRISC Processing Unit

0.2.1.3. RISC-V Processing Unit

0.2.2. Open Source Software

0.2.2.1. MSP430 GNU Compiler Collection

0.2.2.2. OpenRISC GNU Compiler Collection

0.2.2.3. RISC-V GNU Compiler Collection

0.2. RISC-V ISA

0.2.1. ISA Bases

0.2.2.1. RISC-V 32

0.2.2.2. RISC-V 64

0.2.2.3. RISC-V 128

0.2.2. ISA Extensions

0.2.2.1. Base Integer Instruction Set

RV32I : Base Integer Instruction Set (32 bit)

RV32I	31:25	24:20	19:15	14:12	11:7	6:0
LUI RD, IMM	IIIIIII	IIIII	IIIII	III	RD4:0	0110111
AUPIC RD, IMM	IIIIIII	IIIII	IIIII	III	RD4:0	0010111
JAL RD, IMM	IIIIIII	IIIII	IIIII	III	RD4:0	1101111
JALR RD,RS1,IMM	IIIIIII	IIIII	RS14:0	000	RD4:0	1101111
BEQ RS1,RS2,IMM	IIIIIII	RS24:0	RS14:0	000	IIIII	1100011
BNE RS1,RS2,IMM	IIIIIII	RS24:0	RS14:0	001	IIIII	1100011
BLT RS1,RS2,IMM	IIIIIII	RS24:0	RS14:0	100	IIIII	1100011
BGE RS1,RS2,IMM	IIIIIII	RS24:0	RS14:0	101	IIIII	1100011
BLTU RS1,RS2,IMM	IIIIIII	RS24:0	RS14:0	110	IIIII	1100011
BGEU RS1,RS2,IMM	IIIIIII	RS24:0	RS14:0	111	IIIII	1100011
LB RD, RS1	IIIIIII	IIIII	RS14:0	000	RD4:0	0000011
LH RD, RS1	IIIIIII	IIIII	RS14:0	001	RD4:0	0000011
LW RD, RS1	IIIIIII	IIIII	RS14:0	010	RD4:0	0000011
LBU RD, RS1	IIIIIII	IIIII	RS14:0	100	RD4:0	0000011
LHU RD, RS1	IIIIIII	IIIII	RS14:0	101	RD4:0	0000011
SB RS2,RS1	IIIIIII	RS24:0	RS14:0	000	IIIII	0100011
SH RS2,RS1	IIIIIII	RS24:0	RS14:0	001	IIIII	0100011
SW RS2,RS1	IIIIIII	RS24:0	RS14:0	010	IIIII	0100011
ADDI RD,RS1,IMM	IIIIIII	IIIII	RS14:0	000	RD4:0	0010011
SLTI RD,RS1,IMM	IIIIIII	IIIII	RS14:0	010	RD4:0	0010011
SLTIU RD,RS1,IMM	IIIIIII	IIIII	RS14:0	011	RD4:0	0010011
XORI RD,RS1,IMM	IIIIIII	IIIII	RS14:0	100	RD4:0	0010011
ORI RD,RS1,IMM	IIIIIII	IIIII	RS14:0	110	RD4:0	0010011
ANDI RD,RS1,IMM	IIIIIII	IIIII	RS14:0	111	RD4:0	0010011
SLLI RD,RS1,IMM	0000000	IIII	RS14:0	001	RD4:0	0010011

RV32I	31:25	24:20	19:15	14:12	11:7	6:0
SRLI RD,RS1,IMM	0000000	IIII	RS14:0	101	RD4:0	0010011
SRAI RD,RS1,IMM	0100000	IIII	RS14:0	101	RD4:0	0010011
ADD RD,RS1,RS2	0000000	RS24:0	RS14:0	000	RD4:0	0110011
SUB RD,RS1,RS2	0100000	RS24:0	RS14:0	000	RD4:0	0110011
SLL RD,RS1,RS2	0000000	RS24:0	RS14:0	001	RD4:0	0110011
SLT RD,RS1,RS2	0000000	RS24:0	RS14:0	010	RD4:0	0110011
SLTU RD,RS1,RS2	0000000	RS24:0	RS14:0	011	RD4:0	0110011
XOR RD,RS1,RS2	0000000	RS24:0	RS14:0	100	RD4:0	0110011
SRL RD,RS1,RS2	0000000	RS24:0	RS14:0	101	RD4:0	0110011
SRA RD,RS1,RS2	0100000	RS24:0	RS14:0	101	RD4:0	0110011
OR RD,RS1,RS2	0000000	RS24:0	RS14:0	110	RD4:0	0110011
AND RD,RS1,RS2	0000000	RS24:0	RS14:0	111	RD4:0	0110011
FENCE PRED,SUCC	0000PPP	PSSSS	00000	000	00000	0001111
FENCE.I	0000P00	00000	00000	001	00000	0001111

RV64I : Base Integer Instruction Set (64 bit)

RV64I	31:25	24:20	19:15	14:12	11:7	6:0
LWU RD, RS1	IIIIIII	IIIII	RS14:0	110	RD4:0	0000011
LD RD, RS1	IIIIIII	IIIII	RS14:0	011	RD4:0	0000011
SD RD, RS1,RS2	IIIIIII	RS24:0	RS14:0	011	IIIII	0000011
SLLI RD, RS1,IMM	0000000	IIIII	RS14:0	001	RD4:0	0010011
SRLI RD, RS1,IMM	0000000	IIIII	RS14:0	001	RD4:0	0010011
SRAI RD, RS1,IMM	0100000	IIIII	RS14:0	001	RD4:0	0010011
ADDIW RD, RS1	IIIIIII	IIIII	RS14:0	000	RD4:0	0011011
SLLIW RD, RS1	0000000	IIIII	RS14:0	001	RD4:0	0011011
SRLIW RD, RS1	0000000	IIIII	RS14:0	101	RD4:0	0011011
SRAIW RD, RS1	0100000	IIIII	RS14:0	101	RD4:0	0011011
ADDW RD, RS1,RS2	0000000	RS24:0	RS14:0	000	RD4:0	0111011
SUBW RD, RS1,RS2	0100000	RS24:0	RS14:0	000	RD4:0	0111011
SLIW RD, RS1,RS2	0000000	RS24:0	RS14:0	001	RD4:0	0111011
SRLW RD, RS1,RS2	0000000	RS24:0	RS14:0	101	RD4:0	0111011
SRAW RD, RS1,RS2	0100000	RS24:0	RS14:0	101	RD4:0	0111011

0.2.2.2. Standard Extension for Integer Multiply and Divide

RV32M : Standard Extension for Integer Multiply and Divide (32 bit)

RV32M	31:25	24:20	19:15	14:12	11:7	6:0
MUL RD,RS1,RS2	0000001	RS24:0	RS14:0	000	RD4:0	0110011
MULH RD,RS1,RS2	0000001	RS24:0	RS14:0	001	RD4:0	0110011
MULHSU RD,RS1,RS2	0000001	RS24:0	RS14:0	010	RD4:0	0110011
MULHU RD,RS1,RS2	0000001	RS24:0	RS14:0	011	RD4:0	0110011
DIV RD,RS1,RS2	0000001	RS24:0	RS14:0	100	RD4:0	0110011
DIVU RD,RS1,RS2	0000001	RS24:0	RS14:0	101	RD4:0	0110011
REM RD,RS1,RS2	0000001	RS24:0	RS14:0	110	RD4:0	0110011
REMU RD,RS1,RS2	0000001	RS24:0	RS14:0	111	RD4:0	0110011

Standard Extension for Integer Multiply and Divide (64 bit)

RV64M	31:25	24:20	19:15	14:12	11:7	6:0
MULW RD,RS1,RS2	0000001	RS24:0	RS14:0	000	RD4:0	0111011
DIVW RD,RS1,RS2	0000001	RS24:0	RS14:0	100	RD4:0	0111011
DIVUW RD,RS1,RS2	0000001	RS24:0	RS14:0	101	RD4:0	0111011
REMW RD,RS1,RS2	0000001	RS24:0	RS14:0	110	RD4:0	0111011
REMUW RD,RS1,RS2	0000001	RS24:0	RS14:0	111	RD4:0	0111011

0.2.2.3. Standard Extension for Atomic Instructions

RV32A : Standard Extension for Atomic Instructions (32 bit)

RV32A	31:25	24:20	19:15	14:12	11:7	6:0
LR.W AQRL,RD,RS1	00010AQRL	00000	RS14:0	010	RD4:0	0101111
SC.W AQRL,RD,RS2,RS1	00011AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOSWAP.W AQRL,RD,RS2,RS1	00001AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOSADD.W AQRL,RD,RS2,RS1	00000AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOSXOR.W AQRL,RD,RS2,RS1	00100AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOOR.W AQRL,RD,RS2,RS1	01000AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOAMD.W AQRL,RD,RS2,RS1	01100AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOMIN.W AQRL,RD,RS2,RS1	10000AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOMAX.W AQRL,RD,RS2,RS1	10100AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOMINU.W AQRL,RD,RS2,RS1	11000AQRL	RS24:0	RS14:0	010	RD4:0	0101111
AMOMAXU.W AQRL,RD,RS2,RS1	11100AQRL	RS24:0	RS14:0	010	RD4:0	0101111

RV64A : Standard Extension for Atomic Instructions (64 bit)

RV64A	31:25	24:20	19:15	14:12	11:7	6:0
LR.D AQRL,RD,RS1	00010AQRL	00000	RS14:0	011	RD4:0	0101111
SC.D AQRL,RD,RS2,RS1	00011AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOSWAP.D AQRL,RD,RS2,RS1	00001AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOSADD.D AQRL,RD,RS2,RS1	00000AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOSXOR.D AQRL,RD,RS2,RS1	00100AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOOR.D AQRL,RD,RS2,RS1	01000AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOAMD.D AQRL,RD,RS2,RS1	01100AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOMIN.D AQRL,RD,RS2,RS1	10000AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOMAX.D AQRL,RD,RS2,RS1	10100AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOMINU.D AQRL,RD,RS2,RS1	11000AQRL	RS24:0	RS14:0	011	RD4:0	0101111
AMOMAXU.D AQRL,RD,RS2,RS1	11100AQRL	RS24:0	RS14:0	011	RD4:0	0101111

0.2.2.4. Standard Extension for Single-Precision Floating-Point

RV32F : Standard Extension for Single-Precision Floating-Point (32 bit)

RV32F	31:25	24:20	19:15	14:12	11:7	6:0
FLW FRD,RS1	IIIIIII	IIIII	FRS1	010	FRD	0000111
FSW FRS2,RS1	IIIIIII	FRS2	FRS1	010	IIIII	0100111
FMADD.S RM,FRD,FRS1,FRS2,FRS3	FRS3_00	FRS2	FRS1	RM	FRD	1000011
FMSUB.S RM,FRD,FRS1,FRS2,FRS3	FRS3_00	FRS2	FRS1	RM	FRD	1000111
FNMSUB.S RM,FRD,FRS1,FRS2,FRS3	FRS3_00	FRS2	FRS1	RM	FRD	1001011
FNMADD.S RM,FRD,FRS1,FRS2,FRS3	FRS3_00	FRS2	FRS1	RM	FRD	1001111

RV32F	31:25	24:20	19:15	14:12	11:7	6:0
FADD.S RM,FRD,FRS1,FRS2,FRS3	0000000	FRS2	FRS1	RM	FRD	1010011
FSUB.S RM,FRD,FRS1,FRS2,FRS3	0000100	FRS2	FRS1	RM	FRD	1010011
FMUL.S RM,FRD,FRS1,FRS2,FRS3	0001000	FRS2	FRS1	RM	FRD	1010011
FDIV.S RM,FRD,FRS1,FRS2,FRS3	0001100	FRS2	FRS1	RM	FRD	1010011
FSGNJ.S FRD,FRS1,FRS2	0010000	FRS2	FRS1	000	FRD	1010011
FSGNJN.S FRD,FRS1,FRS2	0010000	FRS2	FRS1	001	FRD	1010011
FSGNJX.S FRD,FRS1,FRS2	0010000	FRS2	FRS1	010	FRD	1010011
FMIN.S FRD,FRS1,FRS2	0010100	FRS2	FRS1	000	FRD	1010011
FMAX.S FRD,FRS1,FRS2	0010100	FRS2	FRS1	001	FRD	1010011
FSQRT.S FRD,FRS1,FRS2	0101100	00000	FRS1	RM	FRD	1010011
FLE.S FRD,FRS1,FRS2	1010000	FRS2	FRS1	000	FRD	1010011
FLT.S FRD,FRS1,FRS2	1010000	FRS2	FRS1	001	FRD	1010011
FEQ.S FRD,FRS1,FRS2	1010000	FRS2	FRS1	010	FRD	1010011
FCVT.W.S RM,RD,FRS1	1100000	00000	FRS1	RM	FRD	1010011
FCVT.WU.S RM,RD,FRS1	1100000	00010	FRS1	RM	FRD	1010011
FCVT.S.W RM,RD,FRS1	1101000	00000	FRS1	RM	FRD	1010011
FCVT.S.WU RM,RD,FRS1	1101000	00010	FRS1	RM	FRD	1010011
FMV.X.S RD,FRS1	1110000	00000	FRS1	000	RD	1010011
FCLASS.S RD,FRS1	1110000	00000	FRS1	001	RD	1010011
FMV.S.X RD,FRS1	1111000	00000	RS1	000	FRD	1010011

RV64F : Standard Extension for Single-Precision Floating-Point (64 bit)

RV64F	31:25	24:20	19:15	14:12	11:7	6:0
FCVT.L.S RM,RD,FRS1	1100000	00010	FRS1	RM	FRD	1010011
FCVT.LU.S RM,RD,FRS1	1100000	00011	FRS1	RM	FRD	1010011
FCVT.S.L RM,RD,FRS1	1101000	00010	FRS1	RM	FRD	1010011
FCVT.S.LU RM,RD,FRS1	1101000	00011	FRS1	RM	FRD	1010011

0.2.2.5. Standard Extension for Double-Precision Floating-Point

RV32D : Standard Extension for Double-Precision Floating-Point (32 bit)

RV32F	31:25	24:20	19:15	14:12	11:7	6:0
FLW FRD,RS1	IIIIIII	IIIII	FRS1	011	FRD	0000111
FSW FRS2,RS1	IIIIIII	FRS2	FRS1	011	IIIII	0100111
FMADD.D RM,FRD,FRS1,FRS2,FRS3	FRS3_01	FRS2	FRS1	RM	FRD	1000011
FMSUB.D RM,FRD,FRS1,FRS2,FRS3	FRS3_01	FRS2	FRS1	RM	FRD	1000111
FNMSUB.D RM,FRD,FRS1,FRS2,FRS3	FRS3_01	FRS2	FRS1	RM	FRD	1001011
FNMADD.D RM,FRD,FRS1,FRS2,FRS3	FRS3_01	FRS2	FRS1	RM	FRD	1001111
FADD.D RM,FRD,FRS1,FRS2,FRS3	0000001	FRS2	FRS1	RM	FRD	1010011
FSUB.D RM,FRD,FRS1,FRS2,FRS3	0000101	FRS2	FRS1	RM	FRD	1010011
FMUL.D RM,FRD,FRS1,FRS2,FRS3	0001001	FRS2	FRS1	RM	FRD	1010011
FDIV.D RM,FRD,FRS1,FRS2,FRS3	0001101	FRS2	FRS1	RM	FRD	1010011
FSGNJ.D FRD,FRS1,FRS2	0010001	FRS2	FRS1	000	FRD	1010011
FSGNJN.D FRD,FRS1,FRS2	0010001	FRS2	FRS1	001	FRD	1010011
FSGNJX.D FRD,FRS1,FRS2	0010001	FRS2	FRS1	010	FRD	1010011
FMIN.D FRD,FRS1,FRS2	0010101	FRS2	FRS1	000	FRD	1010011
FMAX.D FRD,FRS1,FRS2	0010101	FRS2	FRS1	001	FRD	1010011
FSQRT.D FRD,FRS1,FRS2	0101101	00000	FRS1	RM	FRD	1010011

RV32F	31:25	24:20	19:15	14:12	11:7	6:0
FLE.D FRD,FRS1,FRS2	1010001	FRS2	FRS1	000	FRD	1010011
FLT.D FRD,FRS1,FRS2	1010001	FRS2	FRS1	001	FRD	1010011
FEQ.D FRD,FRS1,FRS2	1010001	FRS2	FRS1	010	FRD	1010011
FCVT.W.D RM,RD,FRS1	1100001	00000	FRS1	RM	FRD	1010011
FCVT.WU.D RM,RD,FRS1	1100001	00010	FRS1	RM	FRD	1010011
FCVT.D.W RM,RD,FRS1	1101001	00000	FRS1	RM	FRD	1010011
FCVT.D.WU RM,RD,FRS1	1101001	00010	FRS1	RM	FRD	1010011
FCLASS.D RD,FRS1	1110001	00000	FRS1	001	RD	1010011

RV64D : Standard Extension for Double-Precision Floating-Point (64 bit)

RV64D	31:25	24:20	19:15	14:12	11:7	6:0
FCVT.L.D RM,RD,FRS1	1100001	00010	FRS1	RM	FRD	1010011
FCVT.LU.D RM,RD,FRS1	1100001	00011	FRS1	RM	FRD	1010011
FCVT.D.L RM,RD,FRS1	1101001	00010	FRS1	RM	FRD	1010011
FCVT.D.LU RM,RD,FRS1	1101001	00011	FRS1	RM	FRD	1010011
FMV.X.D RD,FRS1	1110001	00000	FRS1	000	RD	1010011
FMV.D.X RD,FRS1	1111001	00000	RS1	000	FRD	1010011

0.2.3. ISA Modes

0.2.3.1. RISC-V User

0.2.3.2. RISC-V Supervisor

0.2.3.3. RISC-V Hypervisor

0.2.3.4. RISC-V Machine

0.3. OpenRISC ISA

0.3.1. ISA Bases

0.3.2.1. OpenRISC 32

0.3.2.2. OpenRISC 64

0.3.2.3. OpenRISC 128

0.3.2. ISA Extensions

0.3.3. ISA Modes

0.3.3.1. OpenRISC User

0.3.3.2. OpenRISC Supervisor

0.3.3.3. OpenRISC Hypervisor

0.3.3.4. OpenRISC Machine

0.4. MSP430 ISA

0.4.1. ISA Bases

0.4.2.1. MSP430 32

0.4.2.2. MSP430 64

0.4.2.3. MSP430 128

0.4.2. ISA Extensions

0.4.3. ISA Modes

0.4.3.1. MSP430 User

0.4.3.2. MSP430 Supervisor

0.4.3.3. MSP430 Hypervisor

0.4.3.4. MSP430 Machine

1. METODOLOGY

1.1. Requirements

1.1.1. Structural UML diagrams

1.1.1.1. Class diagram

1.1.1.2. Component diagram

1.1.1.3. Composite diagram

1.1.1.4. Deployment diagram

1.1.1.5. Object diagram

1.1.1.6. Package diagram

1.1.1.7. Profile diagram

1.1.2. Behavioral UML diagrams

1.1.2.1. Activity diagram

1.1.2.2. Communication diagram

1.1.2.3. Interaction diagram

1.1.2.4. Sequence diagram

1.1.2.5. State diagram

1.1.2.6. Timing diagram

1.1.2.7. Use diagram

1.2. Source

1.2.1. MatLab Language

1.2.2. Rust Language

1.3. Model

1.3.1. VHDL

1.3.2. Verilog

1.4. Validation

1.4.1. VHDL

1.4.2. Verilog

1.5. Design

1.5.1. VHDL

1.5.2. Verilog

1.6. Verification

1.6.1. OSVVM-VHDL

1.6.1.1. OSVVM Checker

1.6.1.2. OSVVM Stimulus

1.6.1.3. OSVVM Testbench

1.6.2. UVM-Verilog

1.6.2.1. UVM Agent

1.6.2.2. UVM Driver

1.6.2.3. UVM Enviroment

1.6.2.4. UVM Monitor

1.6.2.5. UVM Scoreboard

1.6.2.6. UVM Sequence

1.6.2.7. UVM Sequencer

1.6.2.8. UVM Subscriber

1.6.2.9. UVM Test

1.6.2.10. UVM Testbench

1.6.2.11. UVM Transaction

2. PROJECTS

A Random Access Memory (RAM) is a computer memory that can be read and changed in any order, typically used to store working data and machine code. A RAM device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory. RAM contains multiplexing and demultiplexing circuitry, to connect the data lines to the addressed storage for reading or writing the entry.

2.1. Random Access Memory for a Processing Unit

2.1.1. Functionality

2.1.1.1. Structure

Core	Module description
`mpsoc_axi4_spram`	Single-Port RAM for AMBA4 AXI-Lite

Core	Module description
`mpsoc_ahb3_spram`	Single-Port RAM for AMBA3 AHB-Lite
`...mpsoc_ram_1r1w`	RAM Wrapper
`.....mpsoc_ram_1r1w_generic`	RAM Generic Module

Core	Module description
`mpsoc_wb_spram`	Single-Port RAM for WishBone
`...mpsoc_wb_ram_generic`	RAM Generic Module

2.1.1.2. Behavior

2.1.2. Interface

2.1.2.1. Constants

2.1.2.2. Signals

2.1.3. Registers

2.1.4. Interruptions

2.2. Random Access Memory for a System on Chip

2.2.1. Functionality

2.2.1.1. Structure

2.2.1.2. Behavior

2.2.2. Interface

2.2.2.1. Constants

2.2.2.2. Signals

2.2.3. Registers

2.2.4. Interruptions

2.3. Random Access Memory for a Multi-Processor System on Chip

2.3.1. Functionality

2.3.1.1. Structure

2.3.1.2. Behavior

2.3.2. Interface

2.3.2.1. Constants

2.3.2.2. Signals

2.3.3. Registers

2.3.4. Interruptions

3. WORKFLOW

3.1. HARDWARE

1. System Level (SystemC/SystemVerilog)

The System Level abstraction of a system only looks at its biggest building blocks like processing units or peripheral devices. At this level the circuit is usually described using traditional programming languages like SystemC or SystemVerilog. Sometimes special software libraries are used that are aimed at simulation circuits on the system level. The IEEE 1685-2009 standard defines the IP-XACT file format that can be used to represent designs on the system level and building blocks that can be used in such system level designs.

2. Behavioral & Register Transfer Level (VHDL/Verilog)

At the Behavioural Level abstraction a language aimed at hardware description such as Verilog or VHDL is used to describe the circuit, but so-called behavioural modeling is used in at least part of the circuit description. In behavioural modeling there must be a language feature that allows for imperative programming to be used to describe data paths and registers. This is the always -block in Verilog and the process -block in VHDL.

A design in Register Transfer Level representation is usually stored using HDLs like Verilog and VHDL. But only a very limited subset of features is used, namely minimalistic always blocks (Verilog) or process blocks (VHDL) that model the register type used and unconditional assignments for the datapath logic. The use of HDLs on this level simplifies simulation as no additional tools are required to simulate a design in Register Transfer Level representation.

3. Logical Gate

At the Logical Gate Level the design is represented by a netlist that uses only cells from a small number of single-bit cells, such as basic logic gates (AND, OR, NOT, XOR, etc.) and registers (usually D-Type Flip-flops). A number of netlist formats exists that can be used on this level such as the Electronic Design Interchange Format (EDIF), but for ease of simulation often a HDL netlist is used. The latter is a HDL file (Verilog or VHDL) that only uses the most basic language constructs for instantiation and connecting of cells.

4. Physical Gate

On the Physical Gate Level only gates are used that are physically available on the target architecture. In some cases this may only be NAND, NOR and NOT gates as well as D-Type registers. In the case of an FPGA-based design the Physical Gate Level representation is a netlist of LUTs with optional output registers, as these are the basic building blocks of FPGA logic cells.

5. Switch Level

A Switch Level representation of a circuit is a netlist utilizing single transistors as cells. Switch Level modeling is possible in Verilog and VHDL, but is seldom used in modern designs, as in modern digital ASIC or FPGA flows the physical gates are considered the atomic build blocks of the logic circuit.

3.1.1. Front-End Open Source Tools

3.1.1.1. Modeling System Level of Hardware

A System Description Language Editor is a computer tool allows to generate software code. A System Description Language is a formal language, which comprises a Programming Language (input), producing a Hardware Description (output). Programming languages are used in computer programming to implement algorithms. The description of a programming language is split into the two components of syntax (form) and semantics (meaning).

SystemVerilog System Description Language Editor

type:

```
git clone --recursive https://github.com/emacs-mirror/emacs

cd emacs
./configure
make
sudo make install
```

3.1.1.2. Simulating System Level of Hardware

A System Description Language Simulator (translator) is a computer program that translates computer code written in a Programming Language (the source language) into a Hardware Description Language (the target language). The compiler is primarily used for programs that translate source code from a high-level programming language to a low-level language to create an executable program.

SystemVerilog System Description Language Simulator

type:

```
git clone --recursive http://git.veripool.org/git/verilator

cd verilator
autoconf
./configure
make
sudo make install
```

```
cd sim/verilog/tests/wb/verilator
source SIMULATE-IT
```

```
cd sim/verilog/tests/ahb3/verilator
source SIMULATE-IT
```

```
cd sim/verilog/tests/axi4/verilator
source SIMULATE-IT
```

3.1.1.3. Verifying System Level of Hardware

A UVM standard improves interoperability and reduces the cost of repurchasing and rewriting IP for each new project or Electronic Design Automation tool. It also makes it easier to reuse verification components. The UVM Class Library provides generic utilities, such as component hierarchy, Transaction Library Model or configuration database, which enable the user to create virtually any structure wanted for the testbench.

SystemVerilog System Description Language Verifier

type:

```
git clone --recursive https://github.com/QueenField/UVM
```

```
cd sim/verilog/pu/riscv/wb/msim
source SIMULATE-IT
```

```
cd sim/verilog/pu/riscv/ahb3/msim
source SIMULATE-IT
```

```
cd sim/verilog/pu/riscv/axi4/msim
source SIMULATE-IT
```

3.1.1.4. Describing Register Transfer Level of Hardware

A Hardware Description Language Editor is any editor that allows to generate hardware code. Hardware Description Language is a specialized computer language used to describe the structure and behavior of digital logic circuits. It allows for the synthesis of a HDL into a netlist, which can then be synthesized, placed and routed to produce the set of masks used to create an integrated circuit.

VHDL/Verilog Hardware Description Language Editor

type:

```
git clone --recursive https://github.com/emacs-mirror/emacs

cd emacs
./configure
make
sudo make install
```

3.1.1.5. Simulating Register Transfer Level of Hardware

A Hardware Description Language Simulator uses mathematical models to replicate the behavior of an actual hardware device. Simulation software allows for modeling of circuit operation and is an invaluable analysis tool. Simulating a circuit's behavior before actually building it can greatly improve design efficiency by making faulty designs known as such, and providing insight into the behavior of electronics circuit designs.

Verilog Hardware Description Language Simulator

type:

```
git clone --recursive https://github.com/steveicarus/iverilog

cd iverilog
sh autoconf.sh
./configure
make
sudo make install
```

```
cd sim/verilog/tests/wb/iverilog
source SIMULATE-IT
```

```
cd sim/verilog/tests/ahb3/iverilog
source SIMULATE-IT
```

```
cd sim/verilog/tests/axi4/iverilog
source SIMULATE-IT
```

VHDL Hardware Description Language Simulator

type:

```
git clone --recursive https://github.com/ghdl/ghdl

cd ghdl
./configure --prefix=/usr/local
make
sudo make install
```

```
cd sim/vhdl/tests/wb/ghdl
source SIMULATE-IT
```

```
cd sim/vhdl/tests/ahb3/ghdl
source SIMULATE-IT
```

```
cd sim/vhdl/tests/axi4/ghdl
source SIMULATE-IT
```

3.1.1.6. Synthesizing Register Transfer Level of Hardware

A Hardware Description Language Synthesizer turns a RTL implementation into a Logical Gate Level implementation. Logical design is a step in the standard design cycle in which the functional design of an electronic circuit is converted into the representation which captures logic operations, arithmetic operations, *control flow, etc. In EDA parts of the logical design is automated using synthesis tools based on the behavioral description of the circuit.*

Verilog Hardware Description Language Synthesizer

type:

```
git clone --recursive https://github.com/YosysHQ/yosys

cd yosys
make
sudo make install
```

VHDL Hardware Description Language Synthesizer

type:

```
git clone --recursive https://github.com/ghdl/ghdl-yosys-plugin
cd ghdl-yosys-plugin
make GHDL=/usr/local
sudo yosys-config --exec mkdir -p --datdir/plugins
sudo yosys-config --exec cp "ghdl.so" --datdir/plugins/ghdl.so
```

3.1.1.7. Optimizing Register Transfer Level of Hardware

A Hardware Description Language Optimizer finds an equivalent representation of the specified logic circuit under specified constraints (minimum area, pre-specified delay). This tool combines scalable logic optimization based on And-Inverter Graphs (AIGs), optimal-delay DAG-based technology mapping for look-up tables and standard cells, and innovative algorithms for sequential synthesis and verification.

Verilog Hardware Description Language Optimizer

type:

```
git clone --recursive https://github.com/YosysHQ/yosys

cd yosys
make
sudo make install
```

3.1.1.8. Verifying Register Transfer Level of Hardware

A Hardware Description Language Verifier proves or disproves the correctness of intended algorithms underlying a hardware system with respect to a certain formal specification or property, using formal methods of mathematics. Formal verification uses modern techniques (SAT/SMT solvers, BDDs, etc.) to prove correctness by essentially doing an exhaustive search through the entire possible input space (formal proof).

Verilog Hardware Description Language Verifier

type:

```
git clone --recursive https://github.com/YosysHQ/SymbiYosys
```

3.1.2. Back-End Open Source Tools

I. Back-End Workflow Qflow for ASICs

type:

```
sudo apt install bison cmake flex freeglut3-dev libcairo2-dev libgsl-dev \
libncurses-dev libx11-dev m4 python-tk python3-tk swig tcl tcl-dev tk-dev tcsh
```

type:

```
git clone --recursive https://github.com/RTimothyEdwards/qflow

cd qflow
./configure
make
sudo make install
```

3.1.2.1. Planning Switch Level of Hardware

A Floor-Planner of an Integrated Circuit (IC) is a schematic representation of tentative placement of its major functional blocks. In modern electronic design process floor-plans are created during the floor-planning design stage, an early stage in the hierarchical approach to Integrated Circuit design. Depending on the design methodology being followed, the actual definition of a floor-plan may differ.

Floor-Planner

type:

```
git clone --recursive https://github.com/RTimothyEdwards/magic

cd magic
./configure
make
sudo make install
```

3.1.2.2. Placing Switch Level of Hardware

A Standard Cell Placer takes a given synthesized circuit netlist together with a technology library and produces a valid placement layout. The layout is optimized according to the aforementioned objectives and ready for cell resizing and buffering, a step essential for timing and signal integrity satisfaction. Physical design flow are iterated a number of times until design closure is achieved.

Standard Cell Placer

type:

```
git clone --recursive https://github.com/rubund/graywolf

cd graywolf
mkdir build
cd build
cmake ..
make
sudo make install
```

3.1.2.3. Timing Switch Level of Hardware

A Standard Cell Timing-Analizer is a simulation method of computing the expected timing of a digital circuit without requiring a simulation of the full circuit. High-performance integrated circuits have traditionally been characterized by the clock frequency at which they operate. Measuring the ability of a circuit to operate at the specified speed requires an ability to measure, during the design process, its delay at numerous steps.

Standard Cell Timing-Analizer

type:

```
git clone --recursive https://github.com/The-OpenROAD-Project/OpenSTA

cd OpenSTA
mkdir build
cd build
cmake ..
make
sudo make install
```

3.1.2.4. Routing Switch Level of Hardware

A Standard Cell Router takes pre-existing polygons consisting of pins on cells, and pre-existing wiring called pre-routes. Each of these polygons are associated with a net. The primary task of the router is to create geometries such that all terminals assigned to the same net are connected, no terminals assigned to different nets are connected, and all design rules are obeyed.

Standard Cell Router

type:

```
git clone --recursive https://github.com/RTimothyEdwards/qrouter

cd qrouter
./configure
make
sudo make install
```

3.1.2.5. Simulating Switch Level of Hardware

A Standard Cell Simulator treats transistors as ideal switches. Extracted capacitance and lumped resistance values are used to make the switch a little bit more realistic than the ideal, using the RC time constants to predict the relative timing of events. This simulator represents a circuit in terms of its exact transistor structure but describes the electrical behavior in a highly idealized way.

Standard Cell Simulator

type:

```
git clone --recursive https://github.com/RTimothyEdwards/irsim

cd irsim
./configure
make
sudo make install
```

3.1.2.6. Verifying Switch Level of Hardware LVS

A Standard Cell Verifier compares netlists, a process known as LVS (Layout vs. Schematic). This step ensures that the geometry that has been laid out matches the expected circuit. The greatest need for LVS is in large analog or mixed-signal circuits that cannot be simulated in reasonable time. LVS can be done faster than simulation, and provides feedback that makes it easier to find errors.

Standard Cell Verifier

type:

```
git clone --recursive https://github.com/RTimothyEdwards/netgen

cd netgen
./configure
make
sudo make install
```

3.1.2.7. Checking Switch Level of Hardware DRC

A Standard Cell Checker is a geometric constraint imposed on Printed Circuit Board (PCB) and Integrated Circuit (IC) designers to ensure their designs function properly, reliably, and can be produced with acceptable yield. Design Rules for production are developed by hardware engineers based on the capability of their processes to realize design intent. Design Rule Checking (DRC) is used to ensure that designers do not violate design rules.

Standard Cell Checker

type:

```
git clone --recursive https://github.com/RTimothyEdwards/magic

cd magic
./configure
make
sudo make install
```

3.1.2.8. Printing Switch Level of Hardware GDS

A Standard Cell Editor allows to print a set of standard cells. The standard cell methodology is an abstraction, whereby a low-level VLSI layout is encapsulated into a logical representation. A standard cell is a group of transistor and interconnect structures that provides a boolean logic function (AND, OR, XOR, XNOR, inverters) or a storage function (flipflop or latch).

Standard Cell Editor

type:

```
git clone --recursive https://github.com/RTimothyEdwards/magic

cd magic
./configure
make
sudo make install
```

II. Back-End Workflow Symbiflow for FPGAs

3.2. SOFTWARE

3.2.1. Compilers

type:

```
sudo apt install autoconf automake autotools-dev curl python3 libmpc-dev \
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf \
libtool patchutils bc zlib1g-dev libexpat-dev
```

3.2.1.1. RISC-V GNU C/C++

type:

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

cd riscv-gnu-toolchain

./configure --prefix=/opt/riscv-elf-gcc
sudo make clean
sudo make

./configure --prefix=/opt/riscv-elf-gcc
sudo make clean
sudo make linux

./configure --prefix=/opt/riscv-elf-gcc --enable-multilib
sudo make clean
sudo make linux
```

3.2.1.2. RISC-V GNU Go

type:

```
git clone --recursive https://go.googlesource.com/go riscv-go
cd riscv-go/src
./all.bash
cd ../..
sudo mv riscv-go /opt
```

3.2.2. Simulators

type:

```
sudo apt install device-tree-compiler libglib2.0-dev libpixman-1-dev pkg-config
```

3.2.2.1. Spike (For Hardware Engineers)

Building Proxy Kernel

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

git clone --recursive https://github.com/riscv/riscv-pk

cd riscv-pk
mkdir build
cd build
../configure --prefix=/opt/riscv-elf-gcc --host=riscv64-unknown-elf
make
sudo make install
```

Building Spike

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

git clone --recursive https://github.com/riscv/riscv-isa-sim

cd riscv-isa-sim
mkdir build
cd build
../configure --prefix=/opt/riscv-elf-gcc
make
sudo make install
```

3.2.2.2. QEMU (For Software Engineers)

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

git clone --recursive https://github.com/qemu/qemu

cd qemu
./configure --prefix=/opt/riscv-elf-gcc \
--target-list=riscv64-softmmu,riscv32-softmmu,riscv64-linux-user,riscv32-linux-user
make
sudo make install
```

4. CONCLUSION

4.1. HARDWARE

```
cd synthesis/yosys
source SYNTHESIZE-IT
```

4.1.1. GSCL 45 nm ASIC

type:

```
cd synthesis/qflow
source FLOW-IT
```

4.1.2. Lattice iCE40 FPGA

type:

```
cd synthesis/symbiflow
source FLOW-IT
```

4.2. SOFTWARE

4.2.1. RISC-V Tests

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

rm -rf tests
rm -rf riscv-tests

mkdir tests
mkdir tests/dump
mkdir tests/hex

git clone --recursive https://github.com/riscv/riscv-tests
cd riscv-tests

autoconf
./configure --prefix=/opt/riscv-elf-gcc/bin
make

cd isa

source ../../elf2hex.sh

mv *.dump ../../tests/dump
mv *.hex ../../tests/hex

cd ..

make clean
```

elf2hex.sh:

```
riscv64-unknown-elf-objcopy -O ihex rv32mi-p-breakpoint rv32mi-p-breakpoint.hex
riscv64-unknown-elf-objcopy -O ihex rv32mi-p-csr rv32mi-p-csr.hex
...
riscv64-unknown-elf-objcopy -O ihex rv64um-v-remw rv64um-v-remw.hex
```

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

spike rv32mi-p-breakpoint
spike rv32mi-p-csr
...
spike rv64um-v-remw
```

4.2.2. RISC-V Bare Metal

type:

```
rm -rf hello_c.elf
rm -rf hello_c.hex

export PATH=/opt/riscv-elf-gcc/bin:${PATH}

riscv64-unknown-elf-gcc -o hello_c.elf hello_c.c
riscv64-unknown-elf-objcopy -O ihex hello_c.elf hello_c.hex
```

C Language:

```c
#include <stdio.h>

int main() {
  printf("Hello QueenField!\n");
  return 0;
}
```

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

spike pk hello_c.elf
```

type:

```
rm -rf hello_cpp.elf
rm -rf hello_cpp.hex

export PATH=/opt/riscv-elf-gcc/bin:${PATH}

riscv64-unknown-elf-g++ -o hello_cpp.elf hello_cpp.cpp
riscv64-unknown-elf-objcopy -O ihex hello_cpp.elf hello_cpp.hex
```

C++ Language:

```cpp
#include <iostream>

int main() {
  std::cout << "Hello QueenField!\n";
  return 0;
}
```

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

spike pk hello_cpp.elf
```

type:

```
rm -rf hello_go.elf
rm -rf hello_go.hex

export PATH=/opt/riscv-elf-gcc/bin:${PATH}
export PATH=/opt/riscv-go/bin:${PATH}

GOOS=linux GOARCH=riscv64 go build -o hello_go.elf hello_go.go
riscv64-unknown-elf-objcopy -O ihex hello_go.elf hello_go.hex
```

Go Language:

```go
package main

import "fmt"
func main() {
  fmt.Println("Hello QueenField!")
}
```

4.2.3. RISC-V Operating System

4.2.3.1. GNU Linux

Building BusyBox

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

git clone --recursive https://git.busybox.net/busybox

cd busybox
make CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig
make CROSS_COMPILE=riscv64-unknown-linux-gnu-
```

Building Linux

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

git clone --recursive https://github.com/torvalds/linux

cd linux
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu-
```

Running Linux

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

qemu-system-riscv64 -nographic -machine virt \
-kernel Image -append "root=/dev/vda ro console=ttyS0" \
-drive file=busybox,format=raw,id=hd0 \
-device virtio-blk-device,drive=hd0
```

4.2.3.2. GNU Hurd

4.2.4. RISC-V Distribution

4.2.4.1. GNU Debian

4.2.4.2. GNU Fedora