MIPS architecture summary

- RISC – reduced instruction set computer, developed by MIPS Technologies in mid 1980s
- One of the simplest RISC architectures, small hardware, low power consumption
- 1 instruction per clock cycle in basic version
- Dual-bus architecture with 32 general purpose registers connected to ALU
- Separate SRAM
- A range of versions
  - 32-bit vs 64-bit data
  - Single cycle CPU vs pipelined CPU
- Recently synthesizable MIPS cores for embedded FPGA applications are being offered by a number of third party vendors

Basic MIPS architecture

- 32-bit instructions, 6-bit opcode (bits 31:26)
- 32 general purpose registers
- RAM data memory
- ALU operands: 3 registers or 2 register + immediate operand

MIPS instruction formats

- Instructions are divided into three types: R, I and J.
- Every instruction starts with a 6-bit opcode.
  - R-type instructions specify three registers, a shift amount field, and a function field;
  - I-type instructions specify two registers and a 16-bit immediate value;
  - J-type contain a 26-bit jump target address.

The following are the three formats:

<table>
<thead>
<tr>
<th>Type</th>
<th>format (bits)</th>
<th>-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>opcode (6)</td>
<td>rs (5) rt (5) rd (5) shamt (5) funct (6)</td>
</tr>
<tr>
<td>I</td>
<td>opcode (6)</td>
<td>rs (5) rt (5) immediate (16)</td>
</tr>
<tr>
<td>J</td>
<td>opcode (6)</td>
<td>address (26)</td>
</tr>
</tbody>
</table>
Sample MIPS instructions

- **ADD $2, $3, $4**
  - R-type ALU instruction
  - Opcode is 0's, rd=2, rs=3, rt=4, func=000010
  - 000000 00011 00100 00010 00000 00010

- **JALR $3**
  - R-type jump instruction
  - Opcode is 0's, rs=3, rt=0, rd=31 (by default), func=00100
  - 00011 00000 01011 000010000000000000

- **ADDI $2, $3, 12**
  - I-type ALU instruction
  - Opcode is 001000, rs=3, rt=2, imm=12
  - 00011 00010 0000000000001100

- **BEQ $3, $4, 4**
  - I-type conditional branch instruction
  - Opcode is 000100, rs=00011, rt=00100, imm=4 (skips next 4 instructions)
  - 000100 00011 00100 000000000000000000

- **SW $2, 128($3); store word**
  - I-type memory address instruction
  - Opcode is 101011, rs=00011, rt=00010, imm=0000000010000000
  - 101011 00011 00010 0000000010000000

- **J 128; jump to address PC <- PC31-28::128::'b00**
  - J-type pseudodirect jump instruction PC <- PC31-28::IR25-0::'b00
  - Opcode is 000010, 26-bit pseudodirect address is 128/4 = 32
  - 000010 00000000000000000000000000000000

Some MIPS assembler instructions don't have direct hardware implementation

- Eg: abs $1, $3
  - Resolved to:
    - bgez $3, pos; branch on greater or equal to 0
    - sub $2, $0, $3
    - j out
    - pos: add $2, $0, $3
    - out: ...

- Eg: rol $2, $3, $4; $3 = $3 rotated left by $4 bits
  - Resolved to:
    - addi $1, $0, 32; load 'b100000 to $1
    - sub $1, $1, $4; $1 = b10000 - $4
    - srl $1, $3, $1; shift right logical variable: $1 = $3 << $1
    - slv $2, $3, $4; shift left logical variable: $2 = $3 << $4
    - or $2, $2, $1; $2 = $2 or $1

Translation from C to MIPS machine code

A for loop in C:

```
for (i=0; i<n; i++) a[i]=b[i]+10;
```

**MIPS code:**

```
for (i=0; i<n; i++) a[i]=b[i]+10:

loop:
  add $6, $5, $2 # $6 = address of b[i]
  lw $7, 0($6) # load b[i] from memory
  add $7, $7, $10 # $7 = b[i]+10
  add $6, $4, $2 # $6 = address of a[i]
  sw $7, 0($6) # store $7 into a[i]
  addi $2, $2, 4 # increment i
  blt $0, $3, loop # branch if index < n (bti- branch if less than)
```
MIPS pipeline stages

- Fetch (F)
  - read next instruction from memory, increment address counter
  - assume 1 cycle to access memory
- Decode (D)
  - read register operands, resolve instruction in control signals, compute branch target
- Execute (E)
  - execute arithmetic/resolve branches
- Memory (M)
  - Load/Store instructions only
  - perform load/store accesses to memory
  - assume 1 cycle to access memory
- Write back (W)
  - write arithmetic results to register file

MIPS pipeline data hazards

- Data hazards
  - Register values “read” in decode, written during write-back
  - Hazard occurs when dependent instruction separated by less than 2 slots
  - Examples:
    - ADD $1,$2,$X (E) ADD $1,$2,$X (M) ADD $1,$2,$X (W)
    - ADD $X,$2,$X (D) ADD $X,$2,$X (D)
    - ADD $X,$2,$X (D) ADD $X,$2,$X (D) ADD $X,$2,$X (D)
  - In most cases, data generated in same stage as data is required [EX]
  - Data forwarding
    - ADD $1,$2,$X (M) ADD $1,$2,$X (W) ADD $1,$2,$X (W)
    - ADD $X,$2,$X (E) ADD $X,$2,$X (E)
    - ADD $X,$2,$X (E) ADD $X,$2,$X (E)
    - ADD $X,$2,$X (E) ADD $X,$2,$X (E)

Load Hazards

- Stalls required when data is not produced in same stage as it is needed for a subsequent instruction
  - Example:
    - LW $2, 0($X) (M)
    - ADD $X, $2 (E)
- When this occurs, insert a “noop” into EX state, stall F and D
  - LW $2, 0($X) (W)
  - NOP (M)
  - ADD $X, $2 (E)
  - Forward from W to E
ELEC6234 Embedded Processor Synthesis  \textcopyright \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textthinspace \textth

### Example of MIPS pipeline operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>add 36, 5, 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add 37, 7, 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add 38, 4, 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add 32, 2, 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blt 32, 3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi 36, 5, 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 7, 0(6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi 7, 7, 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw 7, 0(6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### picoMIPS instruction groups

- **ALU instructions operate on registers only, not RAM**
  - ADD, SUB, AND, OR, XOR, NOR
  - ADDI, SUBI, ANDI, ORI, XORI, NORI
  - no synchronous RAM, no branches
  - MUL, sHR instructions SLL, SRL, SRA (will not be included in our case study)
- **Program flow control**
  - J address - unconditional absolute jump
  - BEQ fR, %s address - conditional relative branch: branch if fR %s == 0
  - Other conditional branches:
    - BNE fR, %2, 0
    - define BGE fR, %2: BGE fR, %2 == BGE %2, %fR
    - define BLO fR, %2: BLO fR, %2 == BLO %2, %fR
- **RAM related instructions**
  - LW %d, [address] - load word from RAM into register %d
  - SW %d, [address] - store word from register %d into RAM
- **Others**
  - eg. NOP: no operation
  - Subroutine calls and returns, stack instructions - not implemented in our case study
- **General approach when synthesising on an FPGA**
  - the decoder should only implement the instructions used by the program stored in the program memory
  - Hardware efficient approach

### picoMIPS — our case study

- **Inspired by MIPS, but not a version of MIPS**
- **Architecture suited for efficient synthesis on FPGAs with synchronous RAM blocks**
- **User specifies the width of the data bus n, typically 8, 16, 32, 64 bits.**
- **User specifies the instruction set to be implemented in the decoder**

### Translating C statements to picoMIPS machine code - if statement

```c
if (a==0) b=1; else b=2;
```

```assembly
; .def a=20 ; a is register 20
; .def b=21 ; b is register 21
; bne %a, %0, Lelse ; if a!=0 goto Lelse, nb. reg %0 is always 0
; load immediate value to b:
; ldi %b, 1 ; load 1 to %b (ADDI %b, %0, 1)
; j Lend ; jump to Lend
Lelse:
; ldi %b, 2 ; actual code: ADDI %b, %0, 2
Lend:
```
Translating C statements to picoMIPS machine code
- while statement

- `i=0; n=10; a=10; while (i<=n){ a += a; i++;}

```picoMIPS
.def i=16
.def n=17
.def a=18
LDI %i,0
LDI %n,10
LDI %a,10
Loop:
  BLO %n,%i, Lend  ; branch to exit loop if Carry set, i.e. n<i
  ADD %a,%a      ; a = a+a
  ADDI %i,%1     ; i = i + 1
  J Loop        ; jump to Loop
Lend: ...
```

Quick exercise:

- Translate the following ‘for’ statement to picoMIPS assembler
  ```picoMIPS
  for(i=0;i<=n;i++):
  .def i=16
  .def n=17
  ...
  ```

Steps in embedded processor synthesis (picoMIPS case study)

- Develop and test ALU with typical operations: ADD, SUB, etc.
- Develop General Purpose Registers (GPR) and test
- Develop a simple Instruction Decoder and test with ALU and GPR
  - In the decoder implement NOP instruction, ALU instructions, LDR
- Develop Program Memory
- Develop Program Counter
- Enhance Instruction Decoder with J and typical conditional branches, e.g. BEQ, BLO etc.
- Basic processor core is functional at this point
  - Possible enhancements:
    - Add data RAM
    - RAM related instructions, eg. LD, ST
    - Develop an I/O port, and implement in decoder IN and OUT instructions
    - Add ALU hardware to support MUL instruction and shift instructions
    - Subroutine calls and returns
- Your coursework assignment:
  - Modify the picoMIPS to implement a prescribed program
  - Write a formal report
  - Challenge: the smallest possible implementation on Altera Cyclone IV